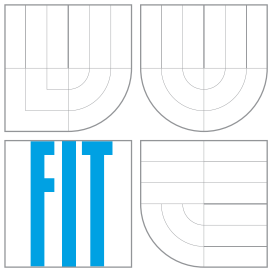
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OBNOVA HESEL V DISTRIBUOVANÉM PROSTŘEDÍ

PASSWORD RECOVERY IN DISTRIBUTED ENVIRONMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. ONDŘEJ KOS

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. RADEK HRANICKÝ

BRNO 2016

Zadání diplomové práce/18213/2015/xkoson00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Kos Ondřej, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Obnova hesel v distribuovaném prostředí**
Password Recovery in Distributed Environment
Kategorie: Bezpečnost

Pokyny:

1. Seznamte se s architekturou a implementací nástroje Wrathion.
2. Nastudujte techniky realizace výpočtů v distribuovaném prostředí.
3. Navrhněte rozšíření nástroje Wrathion, které umožní efektivní běh výpočtů v distribuovaném prostředí s využitím vhodných knihoven (např. Open MPI).
4. Navržené rozšíření implementujte.
5. V distribuovaném prostředí proveďte měření výkonu, zrychlení a škálovatelnosti při výpočtu na různém počtu uzlů.
6. Zhodnoťte dosažené výsledky.

Literatura:

- KIM, Keonwoo. Distributed password cracking on GPU nodes. In: *7th International Conference on Computing and Convergence Technology (ICCCT)*, Seoul: IEEE 2012, s. 647-650. ISBN 978-1-4673-0894-6.
- MENEZES, Alfred J., VAN OORSCHOT, Paul C., VANSTONE, Scott A. Handbook of Applied Cryptography. Boca Raton (Florida): CRC Press 1999. 810 s. ISBN 978-8189836122.
- A další dle dohody s vedoucím.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hranický Radek, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Cílem práce je navržení a implementace frameworku umožňujícího obnovu hesel v distribuovaném prostředí. Úvodní část se tak zaměřuje na analýzu bezpečnosti hesel, technik používaných pro útoky na ně a také představuje metody bránící napadání hesel. Představen je nástroj Wrathion umožňující obnovu hesel za pomocí akcelerace na grafických kartách díky integraci frameworku OpenCL. Provedena je také analýza dostupných prostředí umožňující běh výpočetních úloh na více zařízeních, na jejím základě je pro rozšíření Wrathionu zvolena platforma OpenMPI. Popsány jsou jednotlivé úpravy a přidané komponeny nástroje a celý systém je také podroben experimentům zaměřujícím se na měření škálovatelnosti a náročnosti síťového provozu. Diskutována je také finanční stránka věci z pohledu využitelnosti Wrathionu v cloudovém distribuovaném prostředí.

Abstract

The goal of this thesis is to design and implement a framework allowing password recovery in a distributed environment. The research is therefore focused on analyzing the security of passwords, techniques used for attacks on them and also presents methods preventing attacks on passwords. Described is the Wrathion tool which is allowing password recovery using acceleration on graphic cards through the integration of OpenCL framework. Conducted is also an analysis of available environments providing means to run computing tasks on multiple devices, based on which the OpenMPI platform is chosen for extending Wrathion. Disclosed are various modifications and added components, and the entire system is also subjected to experiments aiming at the measuring of scalability and network traffic performance. The financial side of the use of Wrathion tool is also discussed in terms of its usability in cloud based distributed environment.

Klíčová slova

obnova hesel, hashovací funkce, OpenMPI, OpenCL, distribuované výpočetní systémy

Keywords

password recovery, hash functions, OpenMPI, OpenCL, distributed computation systems

Citace

KOS, Ondřej. *Obnova hesel v distribuovaném prostředí.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Hranický Radek.

Obnova hesel v distribuovaném prostředí

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Radka Hranického. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ondřej Kos
24. května 2016

Poděkování

Rád bych poděkoval Ing. Radkovi Hranickému za pomoc a cenné rady poskytnuté při vypracovávání této práce.

Obsah

1 Úvod **3**

2 Obnova hesel **5**
2.1 Slovníkový útok . . . 5
2.1.1 Vyhledávací tabulka . . . 6
2.1.2 Duhové tabulky . . . 7
2.2 Útok hrubou silou . . . 7
2.3 Metody pro ztížení útoku . . . 7
2.3.1 Sůl . . . 8
2.3.2 Pepř . . . 8
2.3.3 Proměnlivý počet iterací . . . 9
2.3.4 Chybně pravdivé výsledky . . . 9

3 Algoritmizace na GPU **11**
3.1 OpenCL . . . 12
3.2 CUDA . . . 13
3.3 CTM . . . 14

4 Distribuované prostředí **16**
4.1 Frameworky . . . 16
4.1.1 Hadoop . . . 17
4.1.2 BOINC . . . 17
4.1.3 CORBA . . . 17
4.1.4 Oracle Grid Engine . . . 17
4.1.5 Open MPI . . . 18

5 Wrathion **19**
5.1 Architektura . . . 19
5.1.1 Jádro . . . 19
5.1.2 Aplikace . . . 21
5.1.3 Moduly . . . 21
5.1.4 ZIP . . . 21
5.1.5 DOC . . . 21
5.1.6 PDF . . . 22
5.2 Princip . . . 22
5.3 Rozšíření . . . 22

6 Návrh řesení **23**
6.1 Architektura . . . 24
6.1.1 Topologie hvězdy . . . 24
6.1.2 Hierarchická struktura . . . 24
6.1.3 Kombinovaný přístup . . . 25
6.2 Distribuce hesel . . . 26
6.2.1 Polling . . . 26
6.2.2 Addressing . . . 26

7 Implementace **27**
7.1 Arbitr a rezervační server zařízení . . . 27
7.1.1 Třída PasswordArbitrator . . . 27
7.1.2 Třída DeviceReservationServer . . . 29
7.2 Úpravy původního kódu . . . 30
7.3 Třída ThreadedBrutePassGen . . . 32
7.3.1 Úprava rezervačního algoritmu . . . 32
7.3.2 Rozšíření třídy FileFormat . . . 33
7.3.3 Spouštění distribuované varianty . . . 33

8 Experimenty **35**
8.1 Prostředí . . . 35
8.2 Výkonnost ve virtualizovaném clusteru . . . 36
8.2.1 Škálovatelnost . . . 36
8.2.2 Zátěž síťového provozu . . . 38
8.2.3 Finanční zátěž výpočtu hesla . . . 39
8.2.4 Distribuovaná GPU varianta . . . 40
8.3 Zhodnocení experimentů . . . 40
8.4 Další možnosti rozšíření . . . 41

9 Závěr **42**

Literatura **43**

Přílohy **46**
Seznam příloh . . . 47

A Obsah CD **48**

Kapitola 1

Úvod

> In a world in which the total of human knowledge is doubling about every ten years, our security can rest only on our ability to learn
>
> Kevin Mitnick

Bezpečnost našich dat je v dnešní době zmiňována při registraci do téměř každé služby. Narážíme na ukazatele síly hesla, restriktivní podmínky, politiky doby platnosti hesel, a z nich plynoucí pravidelné změny, a také další možné techniky cílící na snížení pravděpodobnosti úspěšného útoku na heslo v podobě tajného klíče. Stejně tak se rádi prezentují výrobci pevných disků a flash pamětí, jaké nativní a automaticky prováděné metody šifrování jejich zařízení podporuje [23], a podobná slova zaznívají i od vývojářů operačních systémů [16]. S trochou nadsázky můžeme říci, že velice brzy mohou být veškerá data, se kterými pracujeme šifrována.

S tímto stavem se však kombinují nepřijemnosti spojené se ztrátou hesla. V případě internetových služeb je zde většinou možnost vyvolat při použití dalšího autentizačního faktoru reset hesla, avšak v případě zašifrovaných lokálně uložených dat tuto možnost nemáme. Zde už nejde pouze o sloupec v tabulce obsahující hash[1] našeho hesla, data jsou v takovémto případě fyzicky změněna šifrováním, a v případě ztráty hesla není jiná možnost, než začít zkoušet veškeré pravděpodobné kombinace hesel.

Uživatelé se zapomenutí jednoho z mnoha užívaných hesel povětšinou brání opětovným používáním jednoho stejného hesla na všech možných místech. Z toho plyne možnost kompromitace zabezpečení využívaných služeb, kdy stačí provést registraci do jediného systému ukládajícího hesla v čisté formě, případně provést útok na službu s nejhorší metodou uchovávání, nebo se obrací na nástroje umožňující generování jedinečných hesel a ukládání přihlašovacích údajů do lokálních databází, opět chráněných dalším heslem (KeePass[2], LastPass[3]).

Ve všech těchto případech vede zapomenutí k problémům. Naštěstí však většinou známe algoritmus, jímž je šifrování realizováno, a taktéž bývá k dispozici kontrolní sekvence, případně ověřovací mechanismus, pomocí které jsme schopni zkontrolovat pravost hesla ještě před samotným dešifrováním celého obsahu, který může v některých případech přesahovat řádově gigabajty dat, a tím pádem by celý proces vygenerování hesla, dešifrování a následná

[1] výsledek hashovací funkce
[2] www.keepass.info/
[3] www.lastpass.com/

kontrola souboru mohly zabrat takové množství času, že by byly nerealizovatelné ani za použití průmyslových výpočetních clusterů. Jen výpočet hashů pomocí dvou průchodů funkce SHA-256 pro všechny kombinace tisknutelných ASCII znaků o délce 10 bajtů (viz tabulku 2.3) by nejvýkonnějšímu superpočítači Tianhe-2 (listopad 2015[4]) zabral přibližně 400 hodin, připočteme-li ke každému výpočtu ještě dešifrovací operaci na daném souboru a výpočet kontrolního součtu, čas lámání v závislosti na velikosti lineárně roste.

Neméně důležitou kategorií pro využití metod obnovy hesel bez jejich znalosti je forenzní analýza a práce bezpečnostních složek. Ve spoustě případů mohou vyšetřovatelé narazit na potenciálně důležitá zašifrovaná data, a osoba, která šifrování prováděla, buďto není k dispozici, nebo odmítá spolupracovat. Je potřeba si uvědomit, že získaná data mohou buďto zamezit kriminální činnosti, nebo přímo vést k odvrácení aktivní hrozby. Je proto důležité mít v takových případech přístup k nástroji, který nám umožní v rozumném čase šifrování prolomit, vezmeme-li v potaz dobrou škálovatelnost, tak za využití veškerého hardware, který máme po ruce.

Tato práce se nejprve zaměřuje na používané metody útoků na hesla a způsoby ochrany, které proti nim byly zavedeny. V další kapitole jsou popsány principy akcelerace výpočtů pomocí grafických karet a dostupné nástroje, v kapitole následující pak metody realizace výpočtů v distribuovaném prostředí. Kapitola 5 se věnuje popisu nástroje Wrathion, který v aktuální verzi umí lámat hesla základních souborových formátů na jednom zařízení, v kapitole 6 jsou pak rozebrány možnosti rozšíření Wrathionu o zvolený framework podporující provádění výpočtů v distribuovaném prostředí ve spojení s algoritmizací na grafické kartě, na což pak navazuje kapitola s popisem implementace zvoleného rozšíření. Kapitola 8 poté představuje navržené experimenty v cloudovém distribuovaném prostředí, a diskutuje jejich výstupy. Zjištěné poznatky a výhody i nevýhody navrženého řešení jsou nakonec diskutovány v závěru.

[4] www.top500.org/lists/2015/11/

Kapitola 2

Obnova hesel

Všeobecně se používají dva pojmy označující v praxi téměř stejnou činnost, a to obnova hesel (angl. password recovery) a lámání hesel (angl. password cracking).

Obnovou hesel rozumíme většinou proces zpětného získání hesla s vědomím osoby, která původní šifrování prováděla. V takovém případě se proces částečně zjednodušuje, jelikož se dá předpokládat, že byť je heslo formálně zapomenuto, jeho autor si může vybavovat alespoň částečně jeho podobu. Tím myslíme délku, použitou sadu znaků, případně i jeho znění, avšak v čisté formě bez obvyklých záměn znaků (tabulka 2.2). Délka hesla zde hraje významnou roli, z tabulky 2.3 je zřejmé, že útokem hrubou silou (viz kapitolu 2.2) na heslo složené z písmen anglické abecedy a čísel (kategorie *alnum*, viz tabulku 2.1), u kterého očekáváme délku mezi 8 a 10 znaky, potřebujeme v průměru vygenerovat 3.76×10^{15} kombinací. Pokud však délku hesla ani přibližně neznáme, a budeme generovat hesla z množiny tisknutelných ASCII znaků o délce 8-16, bude třeba nagenerovat 4.45×10^{31} hesel pro zásah do testovaného hesla, což je více než druhá mocnina množství zmíněné omezené sady o délce 8-10 znaků.

název	**obsažené znaky**
numeric	0-9
alpha	a-z
alnum	0-9a-z
cased alpha	a-zA-Z
cased alnum	0-9a-zA-Z
ASCII-32	0-9a-zA-Z!"#$%&'()*+,-./:;<=>?@[\]^_‘{}~

Tabulka 2.1: Použité značení a symboly zmiňovaných znakových sad

Druhý používaný pojem je lámání hesel - takto označujeme aktivitu, kdy hledáme heslo, o kterém nemáme buďto žádné, nebo pouze velice kusé informace. Dostáváme se tak do situace, kdy musíme využít kompletní spektrum možných použitelných symbolů, které zahrnují malá a velká písmena, číslice a sadu běžných tisknutelných znaků (ASCII-32, tabulka 2.1). Tato množina ještě naroste v případě, že šifrování bylo provedeno osobou kladoucí veliký důraz na bezpečnost, a pro tvorbu klíče byly použity i symboly netisknutelné.

2.1 Slovníkový útok

Jak již napovídá název, pro nalezení hesla se používá předdefinovaný slovník, respektive databáze hesel nebo výsledků hashovacích funkcí nad nimi. V nejčistší formě tedy jde o sez-

A	@, 4
B	8, |3
E	3
H	/-/, # |-|
I	1, !, |
S	5, $

Tabulka 2.2: Příklad záměn znaků v tzv. leet speak [26]

	sada (znaky)					
	numeric	alpha	alnum	cased alpha	cased alnum	ascii-32
délka	(10)	(26)	(36)	(52)	(62)	(95)
3	1.0E+03	1.8E+04	4.7E+04	1.4E+05	2.4E+05	8.6E+05
4	1.0E+04	4.6E+05	1.7E+06	7.3E+06	1.5E+07	8.1E+07
5	1.0E+05	1.2E+07	6.0E+07	3.8E+08	9.2E+08	7.7E+09
6	1.0E+06	3.1E+08	2.2E+09	2.0E+10	5.7E+10	7.4E+11
7	1.0E+07	8.0E+09	7.8E+10	1.0E+12	3.5E+12	7.0E+13
8	1.0E+08	2.1E+11	2.8E+12	5.3E+13	2.2E+14	6.6E+15
9	1.0E+09	5.4E+12	1.0E+14	2.8E+15	1.4E+16	6.3E+17
10	1.0E+10	1.4E+14	3.7E+15	1.4E+17	8.4E+17	6.0E+19
11	1.0E+11	3.7E+15	1.3E+17	7.5E+18	5.2E+19	5.7E+21
12	1.0E+12	9.5E+16	4.7E+18	3.9E+20	3.2E+21	5.4E+23
13	1.0E+13	2.5E+18	1.7E+20	2.0E+22	2.0E+23	5.1E+25
14	1.0E+14	6.5E+19	6.1E+21	1.1E+24	1.2E+25	4.9E+27

Tabulka 2.3: Velikost možného prostoru hesel v závislosti na délce a použité znakové sadě

nam všech hesel, pro získání výsledného hashe odpovídajícího hledanému však musíme projít s daným klíčem v otevřeném formátu (angl. plain-text) stejným procesem, jakým heslo prochází při jeho tvorbě a ukládání. Teoreticky v takovém případě ani nemusí dojít k výraznému zrychlení, záleží na tom, jakým způsobem byl slovník vytvořen. Uvážíme-li cenu I/O[1] operace přečtení hesla z disku a cenu procesorové operace generování následujícího hesla podle zadaného regulárního výrazu, velice pravděpodobně se dostaneme k informaci, že generování hesel je méně nákladné než čtení z existujícího slovníku. Vzhledem k tomu, že tyto výpočty jsou často náročné pro ztížení útoku, používají se takzvané duhové tabulky (angl. rainbow tables)[2].

2.1.1 Vyhledávací tabulka

Vyhledávací tabulky jsou v angličtině označovány jako lookup tables, jde o formu slovníkového útoku, kdy máme uloženy dvojice původního hesla a výsledného hashe. Taková databáze je poté jednoúčelová, protože se dá aplikovat pouze na zjišťování hesel, která prošla stejným procesem, avšak užitečná pro opakování stejné činnosti. Tyto metody jsou často využívány například při hledání hesel do internetových fór, případně pro přístupová hesla Wi-Fi sítí s šifrováním WPA-PSK [22, 21]. Útoky pomocí lookup tables jsou velice rychlé, v praxi

[1] vstup, výstup

[2] www.project-rainbowcrack.com/table.htm

se jedná pouze o porovnání dvou textových řetězců, avšak jsou nepoužitelné, jsou-li do výpočtu hashe hesla zavedeny některé z metod pro ztížení útoku (viz kapitolu 2.3).

2.1.2 Duhové tabulky

Anglické označení rainbow tables nese český ekvivalent v podobě duhových tabulek, jedná se o databázi vstupních řetězců zpracovaných různými hashovacími metodami. Tím dostáváme do rukou nástroj, u kterého stačí jednoduché porovnání hledaného klíče se záznamy v duhové tabulce. Hlavní výhodou duhových tabulek je orientace na čas a rychlost - veškeré hashe jsou vypočítány pouze jednou, a tyto poté můžeme aplikovat na více hesel používajících stejné nebo podobné principy výpočtu. Toto je však kompenzováno obrovským objemem dat. Vezmeme-li v potaz množství hesel (viz tabulku 2.3) a délku výstupu hashovací funkce (například SHA-256 - 64 znaků [17]), pak se dostáváme na desítky gigabajtů dat.

2.2 Útok hrubou silou

Zjednodušeně jde o zkoušení všech možných permutací znaků dané abecedy, dokud nenarazíme na takovou, která odpovídá původnímu heslu. V praxi se pak celý proces skládá ze dvou částí, a to generování hesel a test, zda je heslo správné či nikoliv.

Generátor hesel pracuje nad zadanou abecedou, ať už předem definovanou, nebo určenou pomocí regulárního výrazu. Principielně se jedná o velice jednoduchý automat, který postupně mění znaky na poslední pozici za následující v abecedě, dokud ji nevyčerpá. Poté posouvá na další iteraci předchozí znak a takto na všech úrovních. V okamžiku, kdy jsou všechny pozice obsazeny posledním znakem abecedy, nastavují se opět na první, a přidává se pozice. Takto operuje nejjednodušší generátor všech možných kombinací, v praxi se však můžeme setkat s generátory operujícími podle předem prováděných heuristik. V takovém případě může generátor začínat na hesle o nejčastější délce 8 znaků [8], kombinace provádět podle priority nejčastěji používaných kláves/digramů/trigramů a podobně.

Od generátoru hesel primárně očekáváme, že bude rychlý. I při použití větší logiky při tvorbě zkoušených hesel se však dá předpokládat, že se generátor nestane úzkým hrdlem řešení lámacího nástroje. Nejpomalejší operací v celém řetězci úloh je, i v případě velice rychlých a optimalizovaných algoritmů, výpočet hashů. Pro kontrolu, zda je vygenerované heslo správné, je logicky potřeba provést jeho porovnání s heslem cílovým, které je však u všech používaných metod k dispozici ve formě hashe. Je proto potřeba aplikovat na kandidátní řetězec všechny kroky, kterými prošel hash uložený u souboru, na který útočíme. Musíme tak mít k dispozici informaci o provedených úlohách, která je většinou uvedena buď u definice souborového formátu, případně teoreticky získatelná za pomoci metod reverzního inženýrsví.

2.3 Metody pro ztížení útoku

U webových aplikací, kdy pro ověření správnosti přístupových údajů je potřeba vyplnit formulář a tento odeslat ke zpracování na stanu serveru, se nabízí značné množství metod jak ztížit útok hrubou silou [20]. Mezi ty jednoduché z pohledu implementace a použití lze zařadit následující:

- omezení IP adres - metodou povolených nebo zakázaných adres,
- captcha kódy - opis textu z obrázku, výběr obrázku odpovícajícího textu atd.,

- account lockout - zablokování přístupu v případě opakovaného zadání chybného hesla.

V případě lámání hesel u fyzických souborů však tyto možnosti nemáme, soubor je fyzicky přítomný na disku útočníka, a ten má k dispozici časový prostor a hardwarové prostředky. Omezení počtu špatně zadaných hesel by bylo značně komplikovaně implementovatelné a vždy by jej bylo možné obejít použitím sandbox prostředí a automatizací lámání hesla v něm. Pro tyto důvody je tedy třeba uchýlit se k metodám, které zvýší náročnost výpočtů hesel při útoku hrubou silou, a to i přes to, že se tyto dají částečně považovat za tzv. security through obscurity[3] [25].

2.3.1 Sůl

Sůl v oboru kryptografie označuje nejčastěji náhodný text přidávaný k heslu v průběhu hashování, v praxi je tento pojem často označován anglickým *salt*. Sůl se začala přidávat pro ztížení útoků na hesla za pomocí předvypočítaných dat v podobě lookup/rainbow tables, které jsou užitečné v případě, že výpočet je vždy stejný. Přidáním náhodného řetězce před hashováním hesla jsou pak tyto metody neefektivní, což je i důvod, proč z definice není potřeba brát sůl jako tajnou informaci. Ta potom bývá často uložena přímo vedle hashe v databázové tabulce nebo hlavičce šifrovaného souboru, nebo dokonce jako podřetězec uloženého hashe.

Použití soli však také podléhá pravidlům, přidání krátkého nebo statického řetězce opět nahrává útokům typu lookup/rainbow tables. V případě použití krátké soli, například 3 ASCII znaky dostáváme přibližně 900000 kombinací možných solí. Jedinou limitací předvypočtených tabulek je velikost úložiště, v takovémto případě nám však pro 1MB nejčastějších hesel postačí disk o velikosti 1TB. Je proto důležité použít sůl o dostatečné délce, pro kterou není finančně ani časově výhodné generování vyhledávacích tabulek. Podobnou chybou je již zmíněné použití statického řetězce jako soli pro všechna hesla. Na takovéto heslo se dá nahlížet jako na neosolené, hlavně poté v případě vyzrazení soli (která principielně nemá být tajemstvím) se opět dostáváme do pole vyhledávacích tabulek, pouze rozšířených o znaky soli - množství výsledných hashů odpovídá množství v případě nepoužití soli.

2.3.2 Pepř

Termín kryptografického pepře není zatím pevně definován, principielně se velice podobá soli, avšak s drobnými rozdíly a většinou funguje jako doplněk soli. V praxi je používán převážně dvěma způsoby:

- Tajný klíč - pepř je stejný pro všechna použití, je uložen odděleně od soli, nejčastěji v kódu.
- Neznámá sůl - pepř se mění pro každé použití (závislost na vstupu), není ukládán (je výsledkem výpočtu).

Tyto dvě metody popisují pouze nejčastější praxi, nabízí se i jejich kombinace, rozšíření rotací pevně daných pepřů, závislost na hesle, soli, i rotovaném pepři, atd. Použití obou těchto nejčastějších metod má své výhody i nevýhody, ale v obou případech je zřejmé, že pepř není uložen společně s heslem jako sůl. Zatímco sůl tedy uložíme do patřičného místa společně s hashem daného hesla, pepř se bude nacházet buďto pouze v kódu, čímž narážíme

[3] dosažení bezpečnosti pomocí utajení logiky skrytí citlivých dat

na možnost otevřenosti algoritmu a porušení Kerckhoffsova principu[4], případně security through obscurity, nebo, v případě pepře algoritmicky vypočteného, nebude uložen vůbec a bude pouze přidávat výpočetní složitost hashe konkrétního hesla.

Analogicky u webové aplikace ukládající hash hesla do databáze, hash a sůl budou v databázové tabulce, zatímco pepř, nebo metoda jeho výpočtu, bude skryt v aplikaci a její logice. K hashi a soli se tedy dostaneme pomocí databázového dotazu, případně přes zprostředkující objekt, avšak pro získání pepře je potřeba znát i kód, pomocí kterého byl vygenerován. Ať už jde tedy o klíč či jinou metodu, opět dochází ke zvýšení složitosti a náročnosti celé operace.

Využívání pepře jako zesílení bezpečnosti hesel je logickým vyústěním většiny úniků hesel k webovým službám. U těchto se často používá pro uložení hesel a soli formy relační databáze (SQL), na kterou jsou často cíleny útoky typu SQL injection[5], database authentication bypass[6] a podobné, díky kterým je možné získat data z databáze, avšak přístup ke kódu samotného webserveru, a tím pádem k pepři, bývá složitější.

2.3.3 Proměnlivý počet iterací

Výpočet každého hashe vstupního klíče má určitou časovou náročnost, byť na dnešních výpočetních strojích prakticky zanedbatelnou. Pro uložení hesla je však málokdy provedena pouze jedna iterace hashovacího algoritmu, často jde o tisíce průchodů, a například v případě Lastpass jde o 100000 iterací algoritmu SHA-256 [13]. Pro uživatele, který provádí dešifrování souboru, může být vteřinu trvající výpočet nutný pro ověření pravosti hesla zanedbatelný, avšak v případě útoku tato praktika značně zvyšuje časovou náročnost útoků hrubou silou. Počet iterací pak ani nemusí být předem znám nebo dán specifikací, může být generován z parametrů plain-text hesla podobně, jako sůl nebo pepř, a logika za jeho výpočtem je pak skryta v aplikaci.

2.3.4 Chybně pravdivé výsledky

Známé jako false positives, od hashovací funkce se podle její definice očekává následující:

- jednoduchost výpočtu hashe pro jakákoliv vstupní data,
- jednosměrnost funkce - nemožnost výpočtu původních dat z jejich hashe,
- nemožnost pozměnit data tak, aby jejich hash zůstal stejný,
- nemožnost vypočítat stejné hashe pro dvě různé zprávy.

Od hashovací funkce se také očekává, že pro jakýkoliv vstup dává výstup konstantní délky, čtvrtý bod tohoto výčtu je pak velice zranitelný, překročí-li délka vstupní zprávy výstupní délku hashe. V teoretické i praktické rovině je tedy možné, aby dva různé vstupy hashovací funkce *měly* stejný výstup. Tento fakt se může zdát jako problém, avšak nám dává do rukou možnost využít tuto skutečnost pro účely ztížení lámání hesel zavedením tzv. false-positives.

Funguje-li metoda dešifrování na principu zveřejnění výsledného hashe a jeho porovnání před samotným aktem dešifrování veškerých uložených dat a výpočtu kontrolního součtu, mohou false positive hesla přidat v závislosti na množství dat a automatizovanosti procesu

[4] bezpečnost šifrování má záviset pouze na utajení klíče, nikoliv algoritmu
[5] útok na databázi podstrčením vlastního SQL kódu skrz neošetřený vstup
[6] úprava dotazů a podstrčení informace, že uživatel je již přihlášen, přestože přihlášení neproběhlo

obrovskou časovou náročnost. Tato metoda je k vidění u formátu ZIP, kde pravděpodobnost false positive hesla je 1:256, a to díky délce kontrolní sekvence, která zde skýtá pouze jeden bajt. Samotné dešifrování je navíc efektivně realizovatelné pouze na CPU, vzhledem k práci s operační pamětí by tato realizace na GPU byla časově daleko náročnější.

Kapitola 3

Algoritmizace na GPU

Grafické karty původně sloužily pouze jako slave[1] zařízení procesoru, kdy data byla odesílána grafické kartě k vykreslení na zobrazovacím zařízení, monitoru. S velkým rozvojem v oblasti vykreslování pomocí počítače, hlavně pak počítačových her, rostla potřeba kvalitního grafického zobrazení a tím pádem větší datový tok mezi CPU a grafickou kartou. Tyto kroky vedly ke snížení množství informací které CPU potřebovalo GPU čipu sdělit pro zobrazení scény, tím pádem zjednodušení instrukcí z kompletního obrazu na sérii polygonů a vlastností jak mají být vykresleny. K tomu přibyla potřeba zpětné analýzy dat vypočtených a vykreslených pomocí grafické karty pro případnou úpravu dalších snímků.

Grafické karty jako hardwarová jednotka jsou tak dnes navrženy pro paralelní zpracování obrovského množství dat ve formě obrazových bodů pomocí jedné instrukce, tedy SIMD - single instruction, multiple data. Principielně se dnes jedná o samostatné počítače s velkým množstvím výpočetních jader, která sice výkonem neodpovídají jádrům procesorovým, ale jejich síla tkví v již zmíněném množství a absenci režie řízení operačního systému. Citelný rozdíl je viditelný i v instrukční sadě procesorů a grafických karet. Nejnovější grafické karty společnosti NVIDIA osazené výpočetním jádrem Maxwell používají instrukční sadu o 112 příkazech [18], zatímco procesory značky Intel nejnovější série Core-i7 pracují s instrukční sadou o cca 512 instrukcích [11].

Paralelní programování na CPU je běžná záležitost, je však potřeba si uvědomit, že běžný procesor může reálně zvládat pouze tolik aktivit najednou, kolik má k dispozici jader a vláken. V dnešní době je typický osobní počítač osazen procesorem se čtyřmi nebo šest jádry, reálně pak můžeme počítat se šestnácti paralelními vlákny. Pro využití v plné síle je však potřeba některé z těchto zdrojů uvolnit pro běh operačního systému, přerušení, vstupně/výstupní operace atd. Ze dvanácti souběžně běžících procesů je pak ukrojena část výkonu, správou procesů a paměti pak celý paralelní běh rozharmonizován. Architektura CPU je také výsledkem vývoje udaného jeho hlavní náplní práce, a to zpracovávání dlouhých proudů instrukcí nad jednoduchými daty.

Pro výpočet běžných procesorových úloh je práce na grafickém čipu nepraktická, řežie spojená se zavedením programu na grafickou kartu je vyšší než u procesoru, a každé výpočetní jádro má také omezené paměťové možnosti, pro velké množství jader je k dispozici podstatně méně operační paměti, než u CPU. Pro porovnání, high-end grafická karta NVIDIA TITAN X je osazena 12GB operační paměti a využívá 3072 výpočetních jader, dostáváme tedy 4MB operační paměti na jádro, zatímco procesor střední třídy Intel Core i-5

[1] master/slave model, kdy master zařízení přebírá kontrolu nad slave zařízením, a toto poté provádí zadané úlohy

o čtyřech fyzických jádrech s 8GB operační paměti při "spravedlivém" rozdělení získá 2GB na jádro, tedy 512 krát více. U programování pro grafické karty je obecně práce s pamětí komplikovanější; například realokace paměťového prostoru není standardem OpenCL podporována, dynamická práce s pamětí tak není použitelná a musíme se spokojit se staticky deklarovanými proměnnými.

Potenciál grafických kartet je dnes již využíván ve velké škále profesionálních nástrojů které nemají s počítačovou grafikou na první pohled nic společného. Velké využití mají díky principu práce nástroje pro tvorbu simulací a výpočty založené na lineární algebře, například nástroj Matlab nabízí nativní podporu GPGPU[2] výpočtů pro grafické karty pracující na technologii CUDA[3], stejně tak simulační nástroje ANSYS [19]. Pro mnoho aktivit je potřeba přepsat celý výpočetní algoritmus aby byl zpracovatelný grafickou kartou, tato práce však může u vhodných algoritmů způsobit až stonásobné zrychlení běhu. I po zohlednění faktorů jako je cena hardwaru navíc a vyšší spotřeba celého zařízení je cena za provedenou operaci v závislosti na čase podstatně nižší.

3.1 OpenCL

OpenCL je framework pro programování nad heterogenní sadou hardwaru, jedná se o otevřený standard původně navržený firmou Apple Inc. který následně převzala pracovní skupina Khronos[4], aktuálně se na něm podílí NVIDIA, Intel, AMD, IBM a Qualcomm. Standard poskytuje rozhraní pro implementaci kódu souběžně pracujícího a komunikujcího skrz platfromy CPU, GPU, FPGA[5] a DSP[6], vznikl v roce 2009 jako reakce na rozšiřující se podporu proprietární technologie CUDA a poptávce po open-source řešení. Standard popisuje čtyři abstraktní modely platformy - model platformy, exekuční model, paměťový model a programovací model.

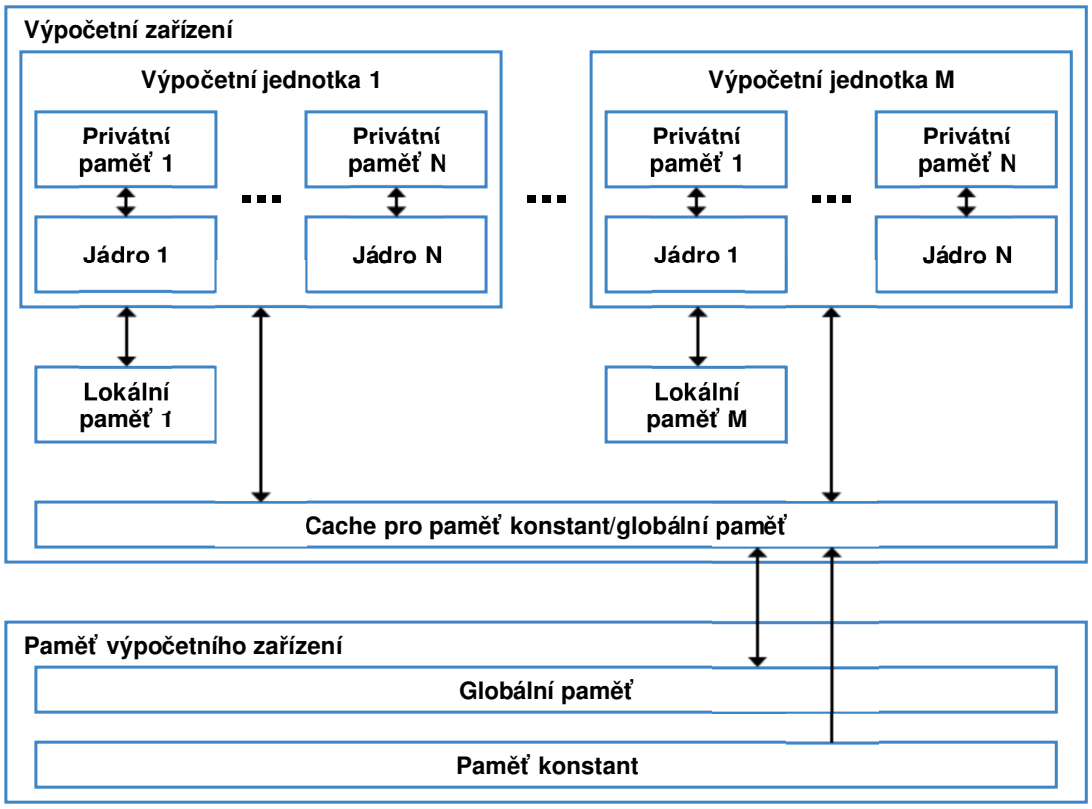
Exekuční model rozděluje programovanou úlohu na dvě části, a to na aplikaci a jádra. Jádra (kernel) jsou repezentací vláken nad jednotlivými výpočetními jádry využívané hardwarové jednotky. Aplikace zajišťuje inicializaci jader, překlad a nahrání programového kódu, a obstarává veškerou potřebnou komunikaci mezi jádry.

Paměťový model popisuje hierarchii použitelného operačního úložiště, tato je rozdělena na čtyři vrstvy (viz obrázek 3.1)

- privátní - každá instance kernelu má vlastní, aplikační část tuto vrstvu nemá implementovánu,
- lokální - společná paměť skupiny jader, aplikace zde nemá přístup,
- paměť konstant - proměnné nastavované aplikací, jádra z této paměti čtou,
- globální - společná paměť celého programu pro čtení a zápis.

Hlavní rozdíl mezi paměťovým modelem běžného procesu pro CPU již byl zmíněn, a to že instance kódu běžícího v jádře GPU nemohou dynamicky alokovat paměť. Tak tomu je i u paměťového modelu OpenCL, zatímco aplikační část může všechny paměťové vrstvy

[2] General-purpose computing on GPU - akcelerace výpočtů na grafické kartě
[3] www.mathworks.com/discovery/matlab-gpu.html
[4] www.khronos.org/
[5] Field-programmable gate array
[6] Digital signal processor



Obrázek 3.1: Paměťový model OpenCL

do nichž má právo zápisu, alokovat dynamicky, jednotlivé kernely pracují pouze se staticky deklarovanými proměnnými.

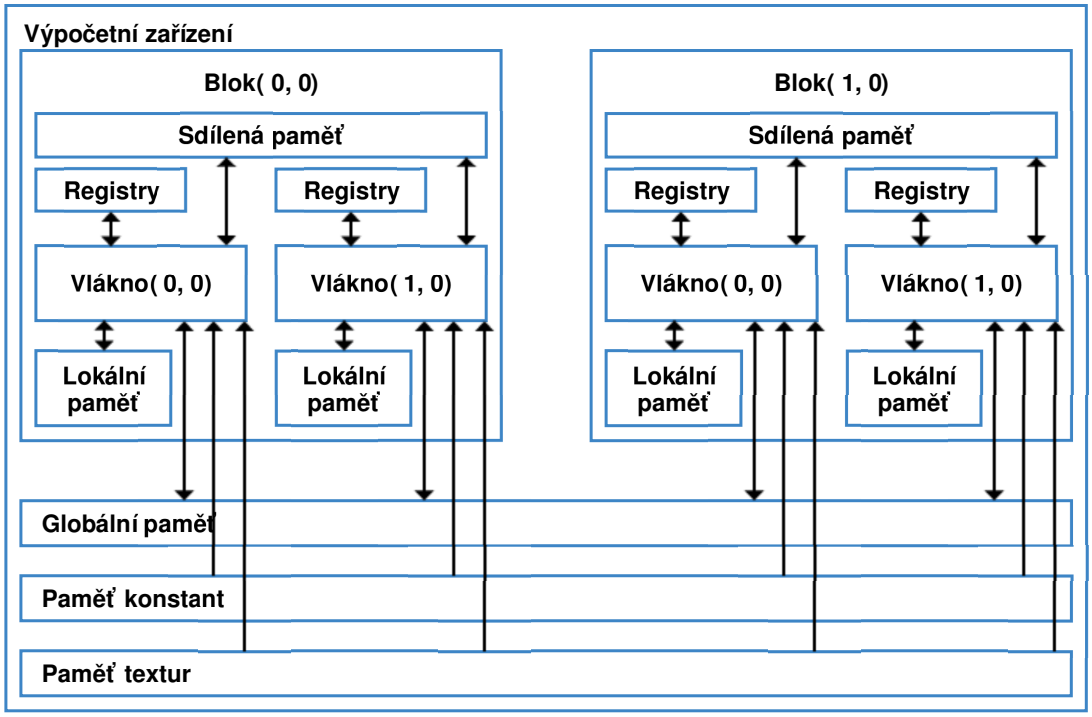
Model platformy pak popisuje celou soustavu zapojených zařízení jako jeden stroj a poskytuje abstrakci nad jednotlivými komponentami, model programovací pak definuje dva typy paralelizmu, a to datový, kdy všechna jádra pracují nad sdílenou sadou dat principem SIMD[7], a typ úlohový, kdy každé jádro zpracovává vlastní kód a vlastní data - MIMD[8].

3.2 CUDA

CUDA (computer unified device architecture) je proprietární technologie firmy NVIDIA šířená pod freeware licencí. CUDA byla představena v roce 2006 a první podporovaná zařízení byly grafické karty z dílny NVIDIA postavené na architektuře jádra Tesla. Principielně je architektura celého virtuálního GPGPU CUDA stroje velice podobná modelu OpenCL, rozdíly jsou především v terminologii a například v paměťové hierarchii, kde CUDA přidává ještě privátní a sdílenou paměť jednotlivých vláken stejného jádra a úroveň společné paměti textur přístupnou pro čtení/zápis pro CPU a pro čtení jednotlivými jádry (viz obrázek 3.2).

Větší podpora ze strany výrobce je také vidět v praktické použitelnosti platformy

[7]Single instruction, multiple data
[8]Multiple instructions, multiple data



Obrázek 3.2: Paměťová hierarchie frameworku CUDA

CUDA. Nativní optimalizace při překladu kódu na GPU spustitelný zajišťuje vyšší rychlost a platforma poskytuje nástroje jako profiler či velice dobře použitelný debugger.

Původním programovacím jazykem využívaným v CUDA bylo upravené C++ rozšířené o vlastní typy se syntaxí odpovídající paralelnímu použití. Ve verzi CUDA 5.2 z roku 2015 je již podpora škály nejvyužívanějších programovacích jazyků, jako jsou například Perl, Python, Java, ale také neméně významných jazyků funkcionálních, a to Haskell a R. Nad technologií CUDA byl také postaven proprietární framework rCUDA, kde *r* zkracuje slovo *remote*, který poskytuje rozhraní pro provádění výpočtů v distribuovaném a vzdáleném prostředí[9].

Porovnání frameworků CUDA a OpenCL na grafických kartách s jádry od Nvidie provedené na univerzitě Delft dopadlo ve prospěch CUDA technologie s přibližně o 30% vyšším výkonem. Kód experimentu byl původně napsán pro CUDA a byl přeložen do OpenCL C, obě verze poté testovány, CUDA zde však těžila z vnitřních optimalizací a faktu, že použitý hardware je pro technologii CUDA optimalizován. Autoři práce se však shodli, že v případě kódových optimalizací OpenCL C kódu testovaného algoritmu by s velkou pravděpodobností bylo dosaženo podobných výsledků jako s CUDA programem [12].

3.3 CTM

Zkratka anglického Close To Metal, jedná se o projekt po NVIDIA druhého největšího výrobce grafických karet ATI (nyní vlastněno AMD). Cílem bylo zpřístupnit instrukční

[9] www.rcuda.net

sadu grafické karty pro vývojáře negrafických aplikací a tím využití dostupného výkonu pro akceleraci výpočtů. Technologie se však dožila pouze beta verze, po které bylo od dalšího vývoje upuštěno, a GPGPU framework z dílny AMD byl napsán od základu znovu pod názvem Stream, nebo také APP (accelerated parallel processing). Po ukončení projektu CTM se však ATI/AMD začalo velmi angažovat ve vývoji OpenCL, které je na grafických kartách z dílny AMD využíváno častěji a dosahuje lepších výsledků. Stream tak zaostává jak za CUDA frameworkem, tak za OpenCL, který je s CUDA po optimalizaci srovnatelný [27].

Kapitola 4

Distribuované prostředí

Pomocí frameworků jako CUDA či OpenCL rozšiřujeme výpočetní sílu zařízení z procesorové jednotky i na periferie v podobě grafické karty či specializovaných čipů. V případě, že máme k dispozici více takovýchto zařízení, realizace některých typů paralelních úloh se dá provádět pro jednotlivé části dat na každém zvlášť. S tím je však spojena velká režie přípravy všech zařízení a manuální rozdělování úloh. Podobně jako OpenCL vytváří abstrakci kterou zapouzdří vícero hardwarových jednotek pod jeden virtuální stroj, frameworky pro distribuované výpočty, jinak také high-performance computing (HPC), vytváří podobnou vrstvu nad sadou jednotlivých fyzických výpočetních strojů. Z jediného centrálního bodu pak můžeme spouštět programový kód, který se paralelně vykonává na všech nám dostupných heterogenních zařízeních a při vhodném a optimalizovaném použití pak můžeme jednotlivé výkony s minimální ztrátou sčítat do jednoho potentního stroje, s nadsázkou superpočítače [5].

V distribuovaném výpočetním prostředí se nejčastěji hovoří o třech kategoriích použití daného systému:

- HPC (high-performance computing) - úlohy vyžadující velké množství výpočetního výkonu na relativně krátkou dobu, běžně hodiny nebo dny, soustředí se na rychlost prováděného úkolu,
- HTC (high-throughtput computing) - úlohy dlouhodobě vyžadující velké množství výpočetního výkonu, řádově měsíce a roky, měrnou jednotkou jsou počet úloh zpracovaných za delší časový úsek,
- MTC (many-task computing) - soustředí se na co nejvyšší počet spočtených úloh za delší časový úsek jako HTC, avšak metriky jsou uváděny proti sekundám jako u HPC (FLOPS, úlohy/s, ...).

4.1 Frameworky

Frameworky pro použití na superpočítačích jsou v naprosté většině proprietárními díly samotných výrobců výpočetních farem, z největších to jsou IBM, HP, Cray a Intel [10], jsou však většinou vázány na speciální hardware a pomineme-li cenu, nejsou prakticky použitelné na běžně dostupném heterogenním mixu počítačů. Pro takové použití je však k dispozici řada užitečných komunitních a open-source projektů.

4.1.1 Hadoop

Hadoop je open-source framework pro distribuované výpočty vyvíjený neziskovou organizací Apache, která si postavila reputaci na nejrozšířenějším webserveru - Apache HTTP server. Platforma Hadoop je zaměřena na operace pracující s velkými objemy dat, takzvané big data (data o objemu který není reálně zpracovatelný na běžném počítači). Framework se skládá z následujících modulů:

- Hadoop common - obsahuje knihovny a nástroje pro práci s big data,
- Hadoop distributed file system - souborový systém navržený pro práci s velkými objemy (řádově petabyty) dat, zaměření na optimalizovanou distribuci po výpočetních uzlech a rychlost a propustnost systému,
- Hadoop YARN - platforma pro správu a plánování výpočetních zdrojů v clusteru.

Nad frameworkem Hadoop staví většina firem vyvíjející big data systémy své proprietární řešení, framework samotný pouze poskytuje aplikační rozhraní a podpůrné nástroje.

4.1.2 BOINC

Systém vyvíjený na Kalifornské univerzitě v Berkeley, je postavený na client-server architektuře a dobrovolném zapojení do výpočtů. Pro zapojení se do výpočetního clusteru je potřeba nainstalovat klientskou část a iniciovat připojení na BOINC server. Server potom připojeným klientským zařízením zadává výpočetní úkoly, jejich výsledky jsou nahrávány zpět na server, kde jsou zavedeny do celého běžícího komplexního výpočtu. Celý systém je postaven na open source řešeních Apache, PHP a MySQL. Jednotlivé výpočetní úlohy jsou označovány jako workunit a systém umožňuje několik optimalizací jejich zpracování. Poskytuje možnost přiřazovat úlohy pouze vybraným operačním systémům, případně zařízením splňujícím zadané hardwarové požadavky. Klientská část pak umožňuje omezit výkon přidělený aplikaci, tím pádem mohou výpočty běžet na pozadí při běžné práci uživatele a pro tuto vlastnost je využíván v komunitních výpočtech. K dispozici je i pro platformu Google Android, mezi nejznámější projekty běžící na platformě BOINC jsou SETI[1], ATLAS[2], Climateprediction[3].

4.1.3 CORBA

Standard vyvinutý pro komunikaci programů psaných v různých jazycích a běžících na různých operačních systémech. Používá jazyk pro definici rozhraní, který je implementován do řady nejpoužívanějších programovacích jazyků (C, C++, COBOL, Java, Lisp, Python, Ruby), je tak možné z imperativního kódu v jazyce C za pomocí aplikačního rozhraní CORBA volat výpočet ve funkcionálním Lispu na jiném stroji, a do výpočtu ještě zakomponovat kód v Pythonu na dalším stroji.

4.1.4 Oracle Grid Engine

Proprietární technologie vycházející z původně open source projektu Sun Grid Engine. Principielně jde o systém podobný BOINC, server pod sebe shromáždí zapojená zařízení,

[1] www.setiathome.berkeley.edu/
[2] www.atlashome.cern.ch/
[3] www.climateprediction.net/

a pomocí fronty na ně distribuuje jednotlivé úlohy. Každé z dostupných zařízení může pracovat na jiném úkolu, nebo také všechny současně na jediném. Systém má robustní řízení výpočtů a plánování.

4.1.5 Open MPI

MPI je zkratka pro *message passing interface*, v překladu tedy rozhraní pro předávání zpráv. Jde o systém, který umožňuje zasílání zpráv jednotlivým procesům, čímž je přijímající strana instruována k volání programového kódu. Jedna z prvních implementací z doby, kdy ještě MPI nebylo definováno, je protokol vzdáleného volání procedur (RPC) z unixových systémů. Standardizované MPI pak představuje protokol pro zasílání zpráv mezi jednotlivými výpočetními stroji a knihovnu poskytující aplikační rozhraní. Open MPI je pak opensource implementace této knihovny vycházející z předchozích úspěšných implementací - PACX-MPI, LAM/MPI, LA-MPI, FT-MPI a Sun CT 6 [7]. Díky komunitě poskytující vlastní hardwarové prostředky pro běh regresních testů je tak celý systém zdarma k dispozici pod open-source licencí. Open MPI navazuje na úspěchy již zmíněných předchozích nástrojů, implementuje standard MPI-2 [14] a architekturou je rozdělen na tři moduly:

- MCA - páteřní komponenta, spravuje všechny vrstvy frameworku,
- Komponentní frameworky - každá funkcionalita Open MPI má svůj vlastní komponentní framework který poté spravuje své moduly,
- Komponenty - samostatné softwarové jednotky poskytující rozhraní pro integraci do distribuovaných komponent.

Z komponentních frameworků pak můžeme zmínit následující:

- Point-to-point management - správa doručování zpráv mezi moduly,
- Byte-transfer-layer layer - komponenta zajišťující doručení dat po síti, obaluje data vyšší úrovně,
- Collective communication - obsluhuje kolektivní MPI operace, poskytuje mezi a mimoprocesovou komunikaci,
- Paralel I/O - moduly implementující rozhraní pro paralelní přístup k souborům a zařízením.

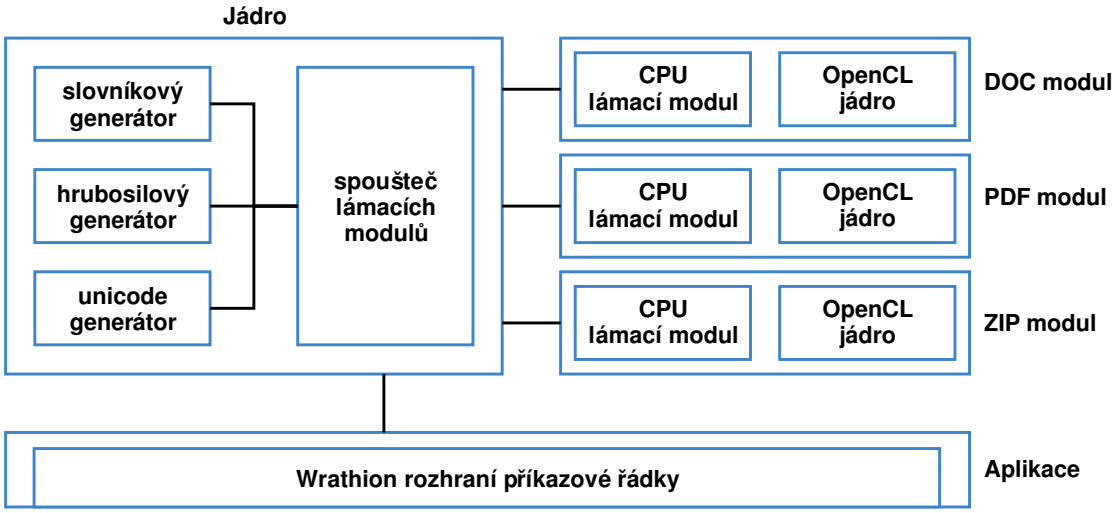
Kapitola 5

Wrathion

Wrathion je nástroj pro lámání hesel, který vznikl na fakultě informačních technologií jako diplomová práce. Díky modulární architektuře, použití open source OpenCL knihovny pro GPU akceleraci a vynikajícím výsledkům v porovnání s ostatními nástroji [24] je předmětem dalšího vývoje a zájmu ze strany bezpečnostních složek.

5.1 Architektura

Architektura nástroje je znázorněna na obrázku 5.1. U obrázku je zřejmé, že nástroj je rozdělen na tři části, a to aplikaci, jádro a moduly.



Obrázek 5.1: Schéma frameworku Wrathion

5.1.1 Jádro

Jádro Wrathionu obsahuje obslužné funkce potřebné pro nahrání a spuštění lámacího procesu. Dále pak obsluhuje načtení souboru, jehož heslo je lámáno a s tím spojenou extrakci signatury souboru, rozpoznání typu, a přípravu důležitých informací z hlavičky souboru,

kterými jsou většinou kontrolní hashe hesla, sůl a případně počet iterací či použitý algoritmus. Součástí jádra jsou také generátory hesel, Wrathion aktuálně poskytuje pouze dva typy:

- slovníkový - prochází odkázaný textový soubor a postupně posílá jednotlivá hesla danému crackovacímu modulu k vyzkoušení,
- bruteforce - generuje všechny možné permutace znaků nad danou abecedou (hesla), omezující podmínkou je zde pouze rozsah délky.

Pro efektivnější generování stojí za zmínku ještě rule-based generátor, kterýžto by pracoval nad předaným regulárním výrazem, tento však není v nástroji Wrathion dosud implementován, a je předmětem plánovaných rozšíření.

Slovníkový generátor již z principu pracuje pouze v CPU režimu, grafická karta nemá fyzický přístup na disk a celkový overhead s implementací logiky jak tento proces realizovat by značně přesáhl jednoduchos instrukce čtení z disku a umístění hesla do sdílené paměti GPU pro vygenerování hashe hesla.

Bruteforce generátor je implementován ve dvou variantách, jednovláknový pracující na principu postupného generování všech možných hesel (demonstrováno na množině {ab}: a, b, aa, ab, ba, bb); a druhý, vícevláknový, pracující na principu rezervace hesel z dostupného stavového prostoru (viz algoritmus 1). Řešení je myšlenkou velice jednoduché a efektivní, každý proces si od řídícího procesu atomickou operací rezervuje daný počet hesel, tím je tedy každému z vláken přiřazeno počáteční heslo (seed) a množství, určující počet postupných hesel vygenerovaných od tohoto bodu. Heslo následující za tímto posledním daného procesu je použito jako seed pro další proces žádající o přidělení sady hesel. Oba bruteforce generátory jsou připraveny pro generování hesel jak na CPU, tak na GPU.

V algoritmu 1 probíhá generování jednotlivých hesel voláním metody `generatePassword`. V případě že proměnná `passwordsLeft`, která představuje počet zbývajících hesel, dosáhne nulové hodnoty, provede se rezervace. `myNextState` představuje následující pozici ve stavovém prostoru hesel, v případě přetečení řádu dochází k rozšíření pole použitého pro výpočty funkcí `increaseArrayState`. Výpočet příští pozice a velikosti rezervace je kritickou sekcí, a tak je uzamčena semaforem `mutex`.

Algoritmus 1: Vícevláknový generátor hesel s rezervacemi [24]

```
1  if passwordsLeft == 0 then
2  |  MutexLock(mutex)
3  |  myNextState = sharedReservation
4  |  sharedReservation = sharedReservation + reservationSize
5  |  MutexUnlock(mutex)
6  |  if reservationSize- stateInt <0 then
7  |  |  increaseArrayState(stateArray, myNextState- stateInt)
8  |  end
9  |  stateInt = myNextState
10 |  passwordsLeft = reservationSize
11 end
12 password = generatePassword(stateArray)
13 passwordsLeft = passwordsLeft- 1
14 return password
```

5.1.2 Aplikace

Aplikace poskytuje rozhraní jádra Wrathionu v podobě command-line utility. Při přerušení crackování předchozího hesla umožňuje při spuštění se stejnými parametry navázat v místě přerušení.

5.1.3 Moduly

Jednotlivé moduly jsou jádrem dynamicky načteny a inicializovány až v okamžiku přečtení typové signatury a připravy inicializačních dat crackeru. Modul pak obsahuje implementaci algoritmu pro ověření pravosti hesla v závislosti na datech z hlavičky šifrovaného souboru, a to ve verzi jak pro CPU, tak pro GPU.

V aktuální verzi Wrathionu (prosinec 2015) je implementována podpora formátů ZIP, DOC a PDF. Popisy jednotlivých algoritmů jsou k dispozici v diplomové práci k implementaci wrathionu [24], u shrnutí se zaměříme na náročnost distribuce potřebných parametrů ke koncovým výpočetním uzlům.

5.1.4 ZIP

Hlavička obsahující informace o komprimaci jsou uvedeny u každého souboru v archivu. Vzhledem k tomu, že u formátu ZIP se nejčastěji setkáváme s false-positive nálezy hesel, je tato informace důležitá a snížení množství umožní více šifrovaných souborů v jednom archivu. Tím je totiž možné ověřit heslo produkující *hit* u jednoho souboru na všech ostatních, a shodují-li se ověřující hashe u všech, nalezli jsme heslo, naopak dojde-li u jediného ze souborů k výpočtu jiného ověřujícího klíče, narazili jsme na false-positive heslo a můžeme pokračovat v crackování. Užitečné proměnné z hlavičky jsou následující:

- typ komprese,
- délka klíče a soli,
- sůl,
- kompresní metoda,
- ověřovací hodnota hesla.

Délka klíče a soli se pohybuje mezi 24 a 48 bajty pro jeden soubor, společně s dalšími informacemi se pak data pro spuštění lámání vejdou do 56 bajtů. Popis komprese a délky klíčů jsou konstantní pro všechny soubory, data navíc pak představují pouze další klíče a soli.

5.1.5 DOC

Proprietární formát společnosti Microsoft, Wrathion podporuje lámání hesel formátu Office Word Document ve verzi šifrování 1.1. V hlavičce máme k dispozici:

- sůl,
- zašifrovanou ověřovací hodnotu,
- zašifrovaný hash ověřovací hodnoty.

Všechny tři parametry jsou 16B dlouhé, pro inicializaci lámání nám tedy stačí rozeslat informaci o délce 48 bajtů.

5.1.6 PDF

Soubory ve formátu PDF nejsou binární, pro čtení hlavičky je tak využita knihovna poppler[1], vyextrahována jsou pak následující data nezbytná pro lámání:

- ověřovací hodnota uživatele,
- verze šifrovacího algoritmu,
- použitá délka šifrovacího klíče,
- revize zabezpečení.

Z pohledu náročnosti distribuovaného výpočtu je pro inicializaci crackeru v případě PDF potřeba přenést na cílové zařízení přibližně 96 bajtů dat. Samotná třída PDFInit však vzhledem ke konstrukci zabírá více než 128 bajtů, z pohledu optimalizace množství zasílaných dat distribučním frameworkem je zde tedy možnost optimalizace.

5.2 Princip

Aplikace načte typovou signaturu lámaného souboru předaného jako argument, podle této je poté vybrán odpovídající modul, který zařídí extrakci parametrů důležitých pro lámání (viz kapitolu 5.1.3). Jádro poté inicializuje příslušný modul a požadovaný generátor hesel (CPU/GPU). Je-li vybrán generátor hesla pracující pouze na CPU, přeloží, nahraje a inicializuje se na GPU obslužný kód pro lámání zvoleného formátu. Do GPU jsou pak předávána vygenerovaná hesla a čteny výsledky použitého algoritmu, dokud nedojde k zásahu správného hesla nebo vyčerpání prostoru kombinací hesel. V případě inicializace generování hesel na GPU pak CPU obstarává pouze přidělování rozsahů hesel jednotlivým jádrům a řízení procesů v případě nálezu hesla, generování a výpočet kontrolního hashe probíhají pouze na grafické kartě.

5.3 Rozšíření

Wrathion je zatím v ranné fázi vývoje, škálovatelný je zatím pouze v rozsahu jenž dovolí jedno koncové zařízení (6 grafických karet) a podpora formátů je také zatím značně omezená. Mezi užitečená rozšíření se dají zařadit:

- Generátor hesel založený na regulárním výrazu - úpravou abecedy se dá dosáhnout výrazného zrychlení, použití záměn znaků (viz tabulku 2.2), aplikace heuristiky, četnosti stisku kláves.
- BitLocker - nástroj pro šifrování disků vyvíjený společností Microsoft [15], spočtený hash pro ověření hesla je přístupný [4].

[1] www.poppler.freedesktop.org/

Kapitola 6

Návrh řesení

Wrathion je dobře modularizován a jako takový na jednom zařízení dobře škálovatelný. Pro rozšíření pomocí Open MPI a dosažení použitelného distribuovaného modelu je potřeba zamyslet se nad dvěmi hlavními změnami oproti původnímu kódu. Knihovna OpenCL bohužel nepodporuje volání instrukcí na vzdáleném grafickém čipu, jako tomu je u rCUDA (viz kapitolu 3.2, je tak potřeba rozšířit framework o možnosti distribuce výpočtů na více uzlů. Kritickou částí je tedy návrh architektury implementovaného distribuovaného prostředí zaměřený na efektivitu výpočtu, a také úprava logiky rezervace hesel, která musí být rozšířena o podporu více zařízení, komunikaci s nimi, a atomičnost této rezervace napříč celým systémem. V následující části budou popsány možnosti řešení distribuované architektury.

Možností volby frameworku pro clusterizaci máme mnoho, budeme-li diskutovat alespoň sadu systémů uvedenou v kapitole 4, zjistíme následující:

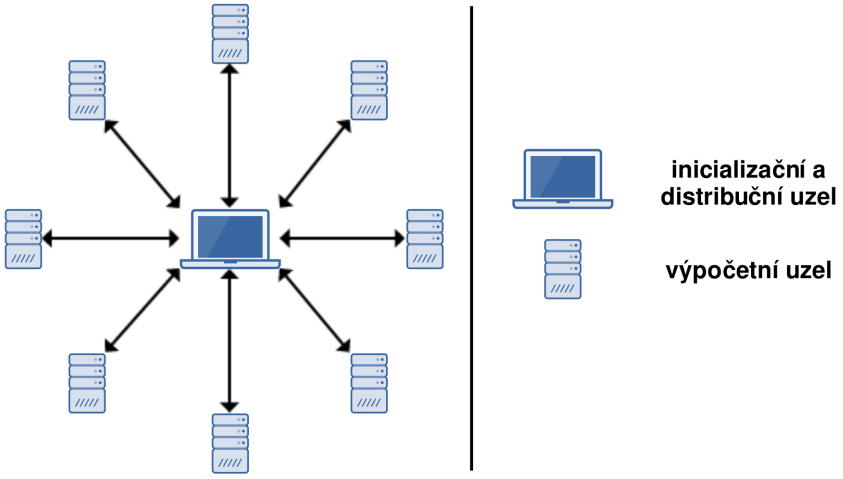
- Hadoop - dá se nasadit i na jiné souborové systémy, než implementovaný HDFS, avšak se ztrátou efektivity, dalším důvodem proč se nezdá použití vhodné je, že Hadoop je primárně vyvíjen pro práci s obrovskými objemy dat, což není případ lámání hesel.
- BOINC - klíčovou částí systému je dobrovolná komunitní aktivita na výpočtech a možnost uvolnit na straně klienta jen zvolenou část výpočetních zdrojů. To by mohlo být užitečné v případě, že bychom nástroj připravovali na šíření výpočtů po internetu, u Wrathionu však počítáme spíše s uzavřenou vnitřní sítí.
- CORBA - architektura je vhodná pro mezijazykovou komunikaci, takové nároky zde však nejsou kladeny.
- Grid - nástroj je dostatečně robustní, jde však o poměrně komplikovanou technologii, implementace není jednoduchou záležitostí, a pro použití ve Wrathionu se dá říct, že jde o příliš potentní framework na jednoduchou úlohu.
- Open MPI - framework je jednoduchý na začlenění do hotového kódu, poskytuje dobré řízení mezivláknové komunikace a dobrou využitelnost v uzavřené síti. Vytváří abstrakci jednoho velice výkonného zařízení zpracovávajícího velký počet vláken, Open MPI se tak jeví jako ideální kandidát na použití ve Wrathionu, rozprostřeme výpočet jednoho hesla na velkou plochu, navíc v kombinaci s technologií CUDA již byly prezentovány slibné výsledky v práci Univerzity Severní Dakoty [3].

6.1 Architektura

Rozšířené jádro wrathionu běžicí na hlavním uzlu vyextrahuje z předaného souboru typovou signaturu a v závislosti na ní data nutná pro lámání hesla. Je zbytečné distribuovat celý šifrovaný soubor na všechny uzly, vzhledem k tomu, že veškeré potřebné infromace zabírají několik desítek bajtů (viz kapitolu 5.1.3), zatímco soubor samotný může zabírat i několik gigabajtů. V závislosti na zvolené distribuční topologii se poté další uzly postarají o delegaci informace na další, za které je zodpovědný. Na všech zařízení je poté potřeba změřit výkon, a v závislost na zjištěných hodnotách spočítat přidělený prostor hesel za zvolenou jednotku času (uvažujme přibližně minutu). Poté v závislosti na zvolené strategii distribuce rozsahů hesel zařídit korektní přiřazení rozsahů odpovědným vláknům a iniciovat pokračování zbytku výpočtu.

6.1.1 Topologie hvězdy

Lámání bude spouštěno ze středového zařízení, kterým bude takový stroj, která má přímé spojení se všemi uzly podílejícími se na výpočtu. Veškerá logika přidělování rozsahů hesel bude prováděna pouze na tomto zařízení, v závislosti na použitém hardwaru pak může být taktéž zapojen do lámacího procesu, nebo nikoliv. V teoretické rovině je jasné, že takto řešený výpočetní cluster není nekonečně škálovatelný, cílem práce pak bude také nalézt limity této topologie.

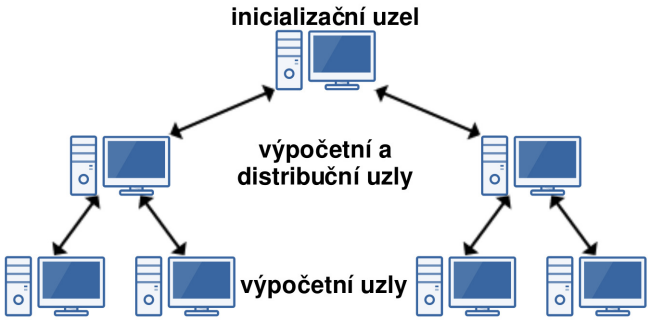


Obrázek 6.1: Distribuovaný výpočet - topologie hvězdy

6.1.2 Hierarchická struktura

Inicializace clusteru bude opět probíhat na jediném zařízení, avšak každý výpočetní uzel bude provádět samotné lámání a zároveň pak také distribuci rozsahů hesel zařízení jemu podřízených, a rezervaci hesel u zařízení nadřazených. U této struktury distribuovaného prostředí bude potřeba sofistikovaněji provádět přerozdělování hesel, logika bude na každém uzlu stejná, musíme ale klást velký důraz na správné nastavení instanciačních proměnných

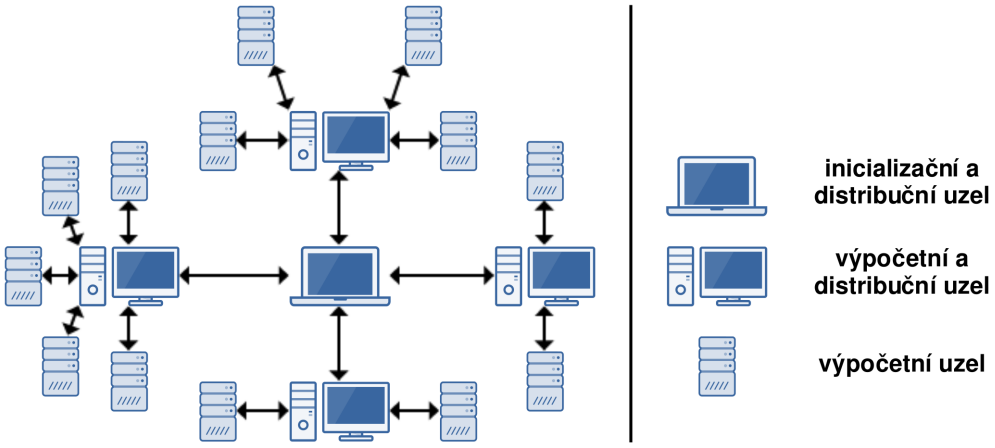
a jejich správné zapracování do algoritmu rezervace v závislosti na výpočetní síle. Rezervace na centrálním uzlu pak bude přiřazovat svým slave uzlům velké bloky hesel, které budou podle stejného principu dále přerozdělovat svým podstromům.



Obrázek 6.2: Distribuovaný výpočet - hierarchická topologie

6.1.3 Kombinovaný přístup

Z dalších možností se nabízí zkombinovat výše uvedené dva principy do jednoho, pro potřeby distribuování výpočtů prováděných Wrathionem připadá v úvahu kombinace stromu a hvězdy, kdy stromová struktura by se omezila na menší množství zařízení a koncové uzly by sloužily jako centrální bod pro hvězdicovou topologii. Druhou možností, na implementaci mnohem jednodušší, je topologie souhvězdí navržená na obrázku 6.3. Centrální bod zde distribuuje data k výpočtu mezi značné množství uzlů metodou point to point, tyto uzly poté pracují na stejném principu a distribuují data dále na své slave uzly. V takovém případě se dá opět pracovat s ne tak výpočetně silným hardwarem pro distribuci dat a použití výkonných strojů na koncových uzlech, jak již bylo navrženo u hvězdicové topologie.



Obrázek 6.3: Distribuovaný výpočet - souhvězdí

6.2 Distribuce hesel

Každý výpočetní uzel bude pracovat nad generováním a zkoušením vlastní sady hesel určené seedem a množstvím hesel následujících. Je proto také třeba řešit otázku jakým principem rozdělovat hesla jednotlivým strojům. Nabízí se dva velice podobné přístupy, a to polling a addressing, teoretická výkonnost a náročnost je velice podobná,

6.2.1 Polling

Celá logika počítání výkonnosti uzlu (a případně jemu podřízených uzlů) musí probíhat na všech strojích vyjma nejvyšší instance. Uzel podle předchozí spočetné sady a z této spočtené průměrné rychlosti lámání vyhodnotí jaké množství hesel si vyžádá od vyšší úrovně. Ta v takovém případě jen "hloupě" odpoví novým seedem a vyžádaným (nebo menším, došel-li této úrovni pool) počtem pokračujících hesel.

6.2.2 Addressing

Uzel, jemuž došel (nebo dochází, chceme-li omezit čekání ve stavu bez hesel) pool hesel pro vyzkoušení hlásí tento stav nadřazenému uzlu, který mu v závislosti na statistice předchozích výpočtů přiřadí další sadu hesel ze sady kterou vlastní. Zpracování statistik a přerozdělování tak probáhá pouze na vyšších úrovních, nikoliv na uzlech koncových, výpočetní náročnost této metody je tedy nižší než u pollingu, a ve větších konfiguracích clusteru se může více projevit v rychlosti a efektivitě.

Kapitola 7

Implementace

Původní kód Wrathionu sice obsahuje značné množství větvení odvolávajícího se na definici překladové proměnné `WRATHION_MPI`, implementace však co se tohoto rozšíření týče zahrnuje pouze nástin návrhu třídy `ClusterBrutePassGen`. Vzhledem k faktu, že zamýšlené řešení by ze značné části s drobnými úpravami kopírovalo rodičovskou třídu `ThreadedBrutePassGen`, byla místo duplikovaného kódu tato adekvátně upravena. Aplikace byla také rozšířena o spouštěcí parametry umožňující úpravy běhu v distribuovaném prostředí, a též byly zavedeny nové třídy obsluhující právě samotnou distribuci jednotlivých úloh mezi uzly clusteru.

7.1 Arbitr a rezervační server zařízení

Pro realizaci samotné distribuce hesel mezi jednotlivé uzly byly připraveny dvě třídy, `PasswordArbitrator` a `DeviceReservationServer`, obě dědící ze třídy `ThreadRunner`. Tato je jednoduchým rozhraním pro implementaci samotných objektů běžících ve vláknech postavených nad knihovnou `pthread`. Práce s vlákny za použití knihoven `boost`[1] nebo nativních vláken ze specifikace C++11 by práci s vlákny zjednodušilo, avšak výkonnost je v těchto slabší, a komplikace by nastaly i v případě překladu statického spustitelného souboru, a to kvůli využívání většího množství dynamických knihoven. Třída `ThreadRunner` je pak navržena jako rozhraní ze kterého dědí obě nově vzniklé třídy obsluhující distribuci, a implementuje základní potřebné operace a to start vlákna jako samostatnou metodu po instanciaci objektu, a vyčkání doběhnutí vlákna pro správné ukončení a uvolnění zdrojů.

7.1.1 Třída PasswordArbitrator

Arbitr pro rozdělování hesel je spuštěn v samostatném vlákně na hlavním MPI uzlu, jeho úlohou je v nekonečné smyčce přijímat příchozí žádosti či informační zprávy od jednotlivých výpočetních uzlů, a spouštět na ně adekvátní obslužné reakce. Z algoritmu 2 je patrné, že cílem po přijetí kontrolní sekvence je co nejrychlejší návrat na instrukci `MPI_Recv` přijímající další data, aby byla obsluha přerozdělování prostorů hesel co nejméně blokující. Tento přístup je hlavním důvodem proč v případě žádosti o vydání nového bloku hesel není tato obsloužena přímo v hlavní smyčce vlákna, ale je instanciován nový objekt třídy `DeviceReservationServer` (viz kapitolu 7.1.2) běžící ve vlastním vlákně.

Kromě obsluhy žádostí o nové balíky hesel pro jednotlivá zařízení spravuje arbitr také informační zprávy o úplném vyčerpání zařízení (heslo nenalezeno), a informaci o tom, že

[1] www.boost.org/

zařízení heslo naopak nalezlo, kdy arbitr poté toto heslo přijme, uloží pro pozdější předání hlavnímu vláknu, a instruuje ukončení ostatních běžících vláken zasláním prázdných balíků při příštích žádostech.

Třída `PasswordArbitrator` pak implementuje ještě obslužné metody využívané v hlavním vlákně, jedná se pouze o funkce zapouzdřující přístup ke stavům privátních proměnných, a to:

- `getCurrentMax` - množství hesel která jsou aktuálně rozřazena mezi zařízení, včetně již vyzkoušených (využíváno pro měření výkonnosti),
- `getIsQuitting` - informace o aktuálním stavu arbitru, dokud není pravdivá, arbitr dále rozděluje hesla, v případě pravdy došlo k nalezení hesla či vyčerpání kompletního rozsahu,
- `getPassFound` - binární informace zda heslo bylo či nebylo nalezeno,
- `getPassword` - předání řetězcové podoby nalezeného hesla.

Objekt arbitra je implementován podle návrhového vzoru jedináčka, chtěným chováním aktuálního návrhu systému je, aby v celém distribuovaném prostředí byl pouze jeden arbitr rozdělující hesla, možnost použití více instancí arbitra je diskutována v kapitole 8.4.

V algoritmu 2, popisujícím chování arbitra, je použita řada metod. `MPI_Recv` a `MPI_Ssend` jsou základní komunikační metody prostředí OpenMPI, kdy první z uvedených zařizuje odeslání dat a druhá jejich příjem. Pro ilustraci algoritmu jsou jejich argumenty zjednodušeny pouze na zařízení se kterým aktuálně funkce komunikuje, a přenášená data označená jako proměnná `payload`, v reálu pak metody pracují s více kontrolními informacemi, jako například přenášený datový typ, velikost, a komunikační kanál. U přijímající metody je pak použita zástupná proměnná `from_any_device`, která signalizuje možnost příjmu dat od kteréhokoliv zařízení, tento faktor je totiž knihovní funkcí omezitelný. Implementace pak může být navržena stylem příjmu od kteréhokoliv zařízení, či specificky od jednotlivých samostatně, v závislosti na návrhu architektury aplikace. Použity jsou také další abstraktní metody provádějící podpůrné úkony, a to:

- `GetDeviceId` - extrakce informace o zařízení které zaslalo data,
- `GetDataFromPayload` - extrakce užitných dat ze zaslaného datového paketu,
- `GetTypeFromPayload` - získání informace zda jde o zprávu řídícího typu.

Algoritmus 2: Hlavní vlákno třídy `PasswordArbitrator`

```
1  while true do
2      MPI_Recv(from_any_device, payload)
3      device = GetDeviceId(payload)
4      data = GetDataFromPayload(payload)
5      type = GetTypeFromPayload(payload)
6      if arbitrator_is_quitting AND type != CONTROL_MESSAGE_TYPE then
           /* arbitr je v ukončovacím režimu -> ukončit procesy na zařízení
              */
7
8          MPI_Ssend(device, TERMINATING_SIGNAL)
9      else if data == NULL AND type == CONTROL_MESSAGE_TYPE then
           /* heslo nalezeno, nebo byl vyčerpán prostor hesel generátoru  */
10
11         MPI_Recv(device, payload)
12         data = GetDataFromPayload(payload)
13         if data == NULL then
               /* heslo nalezeno                                          */
14
15             MPI_Recv(device, payload)
16             password = GetDataFromPayload(payload)
17             password_found = true
18             arbitrator_is_quitting = true
19         else
               /* generátor vyčerpán                                      */
20
21     else
           /* žádost o nový balík hesel                                   */
22         new DeviceReservationServer(device)
23 end
```

7.1.2 Třída DeviceReservationServer

Rezervační server dědí stejně jako třída `PasswordArbitrator` z rozhraní `ThreadRunner` pro jednoduchý běh ve vlákně a jeho kontrolu. Nový objekt této třídy je instanciován po každé žádosti o nový blok hesel, a vlákno jako takové plní pouze tento jednoduchý úkol, po

odeslání dat pracujícímu uzlu pak zaniká.

Algoritmus 3: Hlavní vlákno třídy `DeviceReservationServer`

```
1  elapsed = GetTimeFromLastAllocation()
2  block = GetLastAllocationSize()
3  MutexLock(mutex)
       /* kritická sekce přistupující k vnitřním proměnným arbitra      */
4      new_base = arbitrator.current_password_max
5      new_size = (block / elapsed) * scaling_factor
6      arbitrator.current_password_max += new_size
7  MutexUnlock(mutex)
8  MPI_Ssend(device, new_base)
9  MPI_Ssend(device, new_size)
10 return
```

V algoritmu 3 je opět použita zjednodušená verze metody `MPI_Ssend`, a to stejným způsobem jako v kapitole 7.1.1. Z dalších funkcí jsou pak použity `GetTimeFromLastAllocation`, která z databáze zařízení získá čas uplynulý od poslední alokace balíku hesel daným zařízením, a `GetLastAllocationSize` která stejným způsobem z databáze získá velikost onoho balíku. Vzhledem k faktu, že se jedná o kritickou sekci, a je přistupováno k vnitřním proměnným objektu arbitra, výpočty a přiřazení nových hodnot jsou zamčena semaforem `mutex`, který je inicializován v objektu arbitra.

Po spuštění vlákna voláním třídní metody `StartInternalThread` ve vlákně arbitra dojde k vyhledání odpovídajícího záznamu zařízení z vektoru, ve kterém jsou uloženy. Tento záznam udržuje id zařízení, čas poslední alokace hesel, počáteční bod v prostoru hesel určený tomuto zařízení, a velikost aktuálně přiděleného balíku. Výkonnost uzlu je jednoduše spočtena velikostí posledního bloku hesel ku uplynulému času od minulé alokace, tato konstanta udává počet hesel za vteřinu a je dále použita jako násobek argumentu faktoru škálovatelnosti pro výpočet množství hesel které bude uzel zkoušet v dalším časovém okně. Protože výpočet pracuje s vnitřními hodnotami arbitru, a to aktuální horní hranicí hesel, je tento výpočet chráněn mutexem, který je také inicializován pod objektem arbitra. Samozřejmostí je úprava velikosti nového balíku hesel, pokud vypočtená hodnota nesplňuje minimální, či maximální hranici. Jako maximální je zvolena poměrná část celkového prostoru hesel definovaného abecedou a maximální zkoušenou délkou hesla. Z důvodů minimalizace pravděpodobnosti střetu dvou vláken rezervačního serveru při přístupu do kritické sekce není faktor škálovatelnosti brán jako přesná hodnota (tedy například pět vteřin), ale okolo této je vybrána náhodná v rozmezí $\pm 10\%$ od zvoleného parametru. Úplně identická zařízení by totiž mohla počítat se stejnou rychlostí a tím pádem by mohlo nastávat hladovění na mutexu uzamykajícím změnu stropové hodnoty hesel, blokování by nemuselo nastat vzhledem k přístupu, který je zvolen v metodě `reservePasswords`, ale zavedením určité míry náhodnosti se možnost střetu dvou vláken značně minimalizuje.

7.2 Úpravy původního kódu

Jak bylo již zmíněno dříve, původní kód nástroje Wrathion nebyl přímo uzpůsoben pro běh v distribuovaném prostředí. Hlavní obslužná smyčka byla navržena pro práci pouze na jednom zařízení, nikoliv na více uzlech. Nová podoba řídící části je popsána algoritmy 4 a

5.

Pro správné fungování navrženého řešení distribuované varianty pak byla rozšířena základna argumentů pro spuštění aplikace, a to o hodnotu faktoru škálovatelnosti, který je již popsán v kapitole 7.1.2, a o konfigurační soubor poskytující seznam dostupných zařízení a jejich zařazení - informaci o platformách a OCL zařízeních v případě běhu za pomocí OpenCL kernelů, nebo příznak běhu na procesoru. Pro korektní určení daného uzlu a zaslání správných parametrů se využívá celého názvu daného zařízení získaného metodou `MPI_Get_processor_name`, která vrací stejnou řetězcovou konstantu, jako utilita unixové příkazové řádky `hostname`. Ke každému názvu uzlu je pak za dvojtečkou uvedeno zařízení, nebo jejich seznam, a to ve stejném formátu jako u původního volání wrathionu s parametrem `-map`, tedy následovně:
`<platforma>:<zařízení>[:<GWS[2]>][,<platforma>:<zařízení>[:<GWS>],...]`.

Algoritmus 4: Inicializace hlavního vlákna na řídícím uzlu

```
1  arbitrator = new PasswordArbitrator
2  runArbitrator(arbitrator)
3  format = getFileFormat(input_file_name)
4  formatId = getFileFormatId(format)
5  MPI_Broadcast_Send(formatId)
6  foreach device do
7  |  sendInitData(device)
8  end
9  while arbitrator->isNotQuitting() do
10 |  PrintCrakcingSatistics()
11 end
```

Algoritmus 5: Inicializace hlavního vlákna na pracovním uzlu

```
1  MPI_Broadcast_Receive(formatId)
2  format = receiveInitData()
3  cracker = getCracker(format)
4  runCracker(cracker)
5  while cracker->isRunning() do
6  |  doCracking
7  end
```

Pro lepší rozlišení je knihovní metoda `MPI_Bcast` v algoritmech 4 a 5 rozdělena na odesílající a příjmající, v kódu však takové rozdělení není a určení role je dáno argumenty jejího volání. Metody `sendInitData` a `receiveInitData` jsou nově přidané do rozhraní třídy FileFromat a zajišťují odeslání a přijetí dat extrahovaných ze souboru hlavním vláknem voláním funkce `getFileFormat`, podrobnější popis této úpravy poskytuje kapitola 7.3.2.

Vzhledem k tomu, že kompletní mapování je načteno do kořenového uzlu při inicializaci, a uloženo do vektoru ze kterého jsou jednotlivé záznamy hledány a předávány až na vyžádání konkrétním uzlem, je možné udržovat konfigurační soubor s globálním mapováním veškerých dostupných uzlů, a jejich výběr pro daný běh tak upravovat až konfigurací hostu při spuštení jednotlivých MPI úloh utilitou `mpirun`. Vzorový konfigurační soubor s mapováním uzlů pak

[2] Global Work Size

může vypadat takto:

```
recoveryhost1:cpu
recoveryhost2:0:1:65535
recoveryhost3:1:1,1:2,1:0:32768
recoveryhost4:cpu
```

7.3 Třída ThreadedBrutePassGen

Objekt třídy generátoru je instanciován na každém z MPI zařízení kde spravuje rozdělování hesel mezi jednotlivé běžící procesorová vlákna uzlu, či kernely běžící v jádrech čipů grafických karet.

7.3.1 Úprava rezervačního algoritmu

V původním návrhu aplikace se počítalo s obsluhou lámajících vláken pouze na jednom zařízení, rezervační algoritmus (viz algoritmus 1) postupně přiděluje jednotlivé části neprozkoumaného prostoru hesel pracujícím vláknům, dokud není vyčerpán jeho rozsah definovaný použitou abecedou a délkou hesla (dojde k nastavení flagu `exhausted`). V ideálním případě bychom použili stejný algoritmus v instanci o úrovni výše, tedy na řídícím uzlu clusteru, avšak narážíme na stav, kdy komunikace mezi zařízeními by blokovala alokaci nového bloku, a tedy dané zařízení by bylo po tento čas nevyužito. Z tohoto důvodu řídí rozdělování hesel objekt třídy `PasswordArbitrator` a algoritmus pro rezervaci byl upraven jak je vidět na algoritmu 6.

Algoritmus 6: Upravený rezervační algoritmus

```
1  if myPosition <hostStartPosition then
       /* byl přidělen nový balík hesel                                  */
2      myPosition = hostStartPosition
3  timestamp = GetTimeFromLastAllocation()
4  reservationSize countNewReservationSize(timestamp)
5  if myPosition + (2 * reservationSize) >hostMaxPosition then
       /* rezervace vyčerpá přidělený blok                               */
6      askForPasswords = true
7  updateCounters()
8  if askForPasswords then
       /* rutina přijímající nový blok hesel zatímco cracker běží        */
9      MPI_Ssend(root, NEED_PASSWORDS_SIGNAL)
10     MPI_Recv(root, payload)
11     hostStartPosition = GetDataFromPayload(payload)
12     MPI_Recv(root, payload)
13     hostMaxPosition = GetDataFromPayload(payload)
```

Hlavní úpravou je zde využívání nově zavedených třídních proměnných `hostStartPosition` a `hostMaxPosition` které uměle omezují prostor hesel, ve kterém může zařízení v daný moment operovat. V okamžiku kdy by dvojnásobek rezervovaného balíku pro jedno vlákno či kernel přesáhlo horní limit daného zařízení, dojde k nastavení příznaku dožadujícího se dalších hesel `askForPasswords`. Tento je následně obsloužen zasláním kontrolní sekvence

hlavnímu uzlu po MPI komunikačním kanálu, a to až po nastavení nových proměnných generátoru, který tak může dále pracovat. Zařízení jsou pak zaslány nové hodnoty omezující balík hesel, jehož velikost je založena na rychlosti zpracování minulého a nastavení parametru škálovacího faktoru (jak je popsáno v kapitole 7.1.2). Při další rezervaci podbalíku jedním z vláken či kernelů zařízení pak dochází k alokaci zkoušených hesel již podle nových hodnot. Proměnné `myPosition` a `reservationSize` udržují hodnoty aktuální pozice v prostoru hesel a velikosti aktuálně alokovaného balíku pro dané vlákno.

7.3.2 Rozšíření třídy FileFormat

Pro potřeby vzdálené inicializace objektu třídy `FileFormat` je zapotřebí jednoznačná identifikace jednotlivých formátů. Jako možnosti se nabízí již používané proměnné, a to signatura, řetězcová reprezentace souborového formátu, a nebo přípony souboru. Všechny tyto hodnoty jsou však typu `string` a jejich přenos pomocí OpenMPI komunikace pak představuje větší zátěž pro síťovou vrstvu, a v případě nestandartní přípony o délce jiné než tři znaky také kontrolu délky přenášeného řetězce před přijetím, což platí i pro textovou reprezentaci typu souboru.

Z těchto důvodů byla třída `FileFormat` a z ní vycházející jednotlivé implementace formátových modulů, rozšířena o celočíselnou reprezentaci jednotlivých formátů. Pro vzdálenou inicializaci a výběr správného modulu pro instanciaci objektu třídy `FileFormat` je pak přenesena pouze tato hodnota typu `uint8_t`[3].

Pro potřeby inicializace dat na vzdáleném uzlu bez nutnosti přenosu celého souboru pak bylo rozhraní třídy `FileFormat` rozšířeno o dvě podstatné metody, a to `sendInitData` a `remoteInit`. První z těchto metod je volána pouze na kořenovém OpenMPI uzlu, a to po instanciaci objektu třídy za užití dat ze souboru jehož heslo je hledáno. Principem metody je postupné odeslání všech prvků struktury `.*InitData` (z původních formátů jde o PDFInitData, DOCInitData a ZIPInitData) z hlavního uzlu na všechny uzly podřízené. Druhá z těchto metod (`remoteInit`) je pak volána na všech zapojených lámacích uzlech a obstarává inicializaci struktury s daty a její naplnění přijatými daty.

Pro účely této výměny dat se nabízí použití podpůrných funkcí aplikačního rozhraní OpenMPI pro přenos celých datových struktur, v takovém případě je však potřeba příprava samotného datového typu za pomocí knihony MPI, která zůstává na zodpovědnosti přímo aplikačního programátora, neboť algoritmická serializace by byla značně netriviální. Inicializace struktur `MPI_Datatype` poskytujících toto zapouzdření však vyžaduje znalost knihovny OpenMPI, kterou od osoby vyvíjející modul souborového formátu nutně neočekáváme, proto jsou metody pro odeslání a příjem inicializačních dat psány po krocích zpracování jednotlivých prvků strukury za pomocí předpřipravených obslužných maker. Komunikace v těchto je pak prováděna pouze pomocí jednoducých metod `MPI_Ssend` a `MPI_Recv`.

7.3.3 Spouštění distribuované varianty

Pro korektní běh upravené verze Wrationu na clusteru je logicky nejdůležitější právě použitá verze MPI knihovny. Starší verze varianty OpenMPI trpěly na nekorektní provádění úkonů při vícevláknovém režimu, nedeterministicky docházelo k nepotvrzení odeslání dat (u synchronní metody `MPI_Ssend`), či nekorektnímu příjmu dat funkcí `MPI_Recv`, toto chování bylo pozorováno i ve verzích 1.8.x a 1.9.x které jsou dostupné jako balíčky většiny nejnovějších verzí linuxových distribucí. Při vývoji a experimentech je tak použita verze knihovny 1.10.2,

[3] celočíselná hodnota o velikosti 8 bitů

která těmito neduhy již netrpí, v případě OpenMPI je ale potřeba manuálně zdrojový kód zkompilovat s přepínačem povolujícím běh v režimu `MPI_THREAD_MULTIPLE`[4]. Nabízí se ještě možnost použít variantu `mpich`[5], která je u mnohých distribucí balíčkována již s podporou vícevláknových aplikací, ale s tímto prostředím nebyl wrathion dostatečně testován. Komplikace mohou také nastat při použití komunikace skrz Infiniband[6], komponenta tohoto rozhraní v knihovně OpenMPI není implementována podle thread-safe[7] principů, kvůli čemuž není zaručena korektnost chování, v případě MPICH tyto komplikace nenastávají, neboť komponentu InfiniBand neobsahuje.

Druhou z podmínek úspěšného spuštění je provázání všech zařízení bezheslovým SSH spojením, důvodem je způsob, jakým provádí komponenta ORTE[8] inicializaci. ORTE je jeden ze tří stavebních bloků OpenMPI (spolu s aplikačním rozhraním a vrstvou zajišťující přístup k datům OPAL[9]), a vytváří prostředí pro běh MPI úlohy mezi jednotlivými zařízeními. Spouštěcí utilita `mpirun` vytvoří ssh tunely mezi jednotlivými zařízeními a na těchto spustí ORTE démona[10], který následně nastartuje zbylé komponenty komunikačního stacku, samotný proces propojení uzlů pomocí ssh ale neprobíhá z centrálního řídícího uzlu na ostatní (topologie hvězdy), nýbrž stromovou strukturou [6]. Tato navíc není v případě spuštění identické úlohy nekolikrát po sobě nikdy stejná, ale vytvářena při startu, a z tohoto důvodu musí všechna zapojená zařízení být schopna navázat bezheslové ssh spojení mezi sebou.

[4] Umožňuje korektní směrování jednotlivých MPI komunikačních zpráv mezi jednotlivá běžící vlákna.
[5] www.mpich.org/
[6] www.infinibandta.org/
[7] Přístup k programování jehož výsledný kód je bezpečně spustitelný ve více vláknech
[8] Open Run-Time Environment
[9] Open Portable Access Layer
[10] orted

Kapitola 8

Experimenty

Účinnost a využití nástroje Wrathion byla v porovnání s ostatními nástroji již dostatečně prokázána ve výsledcích měření diplomové práce která tento nástroj představila a hlavně pak článku, který práci fakulty prezentoval [24, 8]. Z těch je zřejmé, že nástroj jako takový je v mnoha případech podstatně rychlejší než konkurenční proprietární systémy, rozšířením o možnost provádění výpočtů v distribuovaném prostředí tak nehledáme zrychlení ve formě více hesel za jednotku času na zařízení, ale spíše pozitivní či negativní vychýlení od křivky ideálního průběhu. Jako křivku ideálního průběhu si můžeme představit takovou, u které pokud provádíme výpočty na naprosto identických zařízeních, kdy pro představu jedno zařízení pracuje rychlostí 100000 hesel za vteřinu, přidáním druhého zařízení do výpočetního clusteru se dostaneme přesně na dvojnásobný počet vyzkoušených hesel, se třetím zařízením na 300000 hesel za vteřinu, atd. V takovémto ideálním případě je tedy počet hesel za jednotku času roven počtu zapojených zařízení, tato křivka je poté uvedena jako reference v grafu 8.1.

8.1 Prostředí

Distribuovaná varianta je vyvíjena převážně za cílem zkrácení doby potřebné pro nalezení hesel použitím všech dostupných výpočetních prostředků. Vzhledem k tomu, že se předpokládá její hlavní využití ze strany bezpečnostních složek, u kterých se předpokládá vlastnictví sady podobných zařízení, ať už fyzického hardware či virtualizovaných počítačů, nad kterými má správce orchestrující spuštění lámacího procesu plnou kontrolu, je potřeba podobné prostředí nasimulovat i pro experimenty zkoumající úspěšnost implementace rozšíření.

Dostupnost dostatečného množství takovýchto fyzických hardwarových prostředků pro ověření všech parametrů důležitých pro distribuovaný výpočet je pro běžného uživatele povětšinou značně limitovaná. Jako řešení se však v dnešní době nabízí pronájem virtualizovaných instancí v cloudovém prostředí. To poskytuje značné množství výhod pro podobné účely:

- jednotlivé stroje jsou dedikovány pouze pro jejich majitele,
- možnosti volby výkonnostních úrovní jednotlivých zařízení,
- vytvoření snapshotu[1] stroje a poté rychlá tvorba jeho kopií,

[1] obraz disku

- poskytují aplikační rozhraní usnadňující ovládání instancí,
- poplatky za pronájem virtuálních strojů jsou účtovány obvykle za hodinu.

Na poli krátko až dlouhodobého pronájmu výpočetních uzlů se angažuje značné množství společností a nabízí nepřeberné množství virtuálních zařízení, od jednoduchých "one-click-ready" zařízení které poskytuje například hosting DigitalOcean[2], až po značně konfigurovatelné systémy cílící převážně na podnikovou sféru, jakou poskytuje Amazon Web Services (AWS)[3]. U AWS stojí za zmínku kromě velice agilní konfigurace také zavedení virtualizovaných GPU zařízení podporující platformu CUDA [2], v porovnání se zvolenou testovací konfigurací jsou však tyto mnohonásobně dražší, jak je diskutováno v kapitole 8.2.3.

Konfigurace zvolená pro testování v cloudovém prostředí DigitalOcean se sestává z několika identických virtuálních počítačů pracujících s dvoujádrovými procesory Intel Xeon Processor E5-2650L v3 [9], 2GB RAM paměti a SSD disku. Testy probíhaly pod linuxovou distribucí Fedora 23 v 64bitové variantě, časová náročnost přípravy primárního výpočetního uzlu (instalace balíčků esenciálních pro překlad wrathionu a kompilace knihovny openmpi) činí přibližně třicet minut, díky možnosti vytvoření snapshotu a instanciaci nového uzlu z uloženého obrazu je však tento proces potřeba absolvovat pouze jednou, a v tomto prostředí tedy i zaplatit pouze za jeden úkon inicializace. V praxi a v reálném prostředí se pak nabízí použití některého z nástrojů pro instalaci na více zařízení (ansible[4], salt[5], puppet[6], ...).

8.2 Výkonnost ve virtualizovaném clusteru

Distribuovaná verze Wrathionu je samozřejmě spustitelná i na jednom zařízení a to včetně veškeré MPI komunikace. Díky tomu je změřitelná režie pro jedno zařízení a tím i poměr výkonnosti oproti běhu wrathionu samostatného. Provedení experimentu pak bylo realizováno postupně na jednom až 25 identických výpočetních uzlech, stejná konfigurace všech zařízení byla zvolena pro jednodušší výpočet poměru výkonu celého clusteru oproti uvažovanému ideálnímu stavu, kdy počet hesel clusteru za vteřinu přímo odpovídá násobku počtu zapojených uzlů a výkonu nedistribuované varianty.

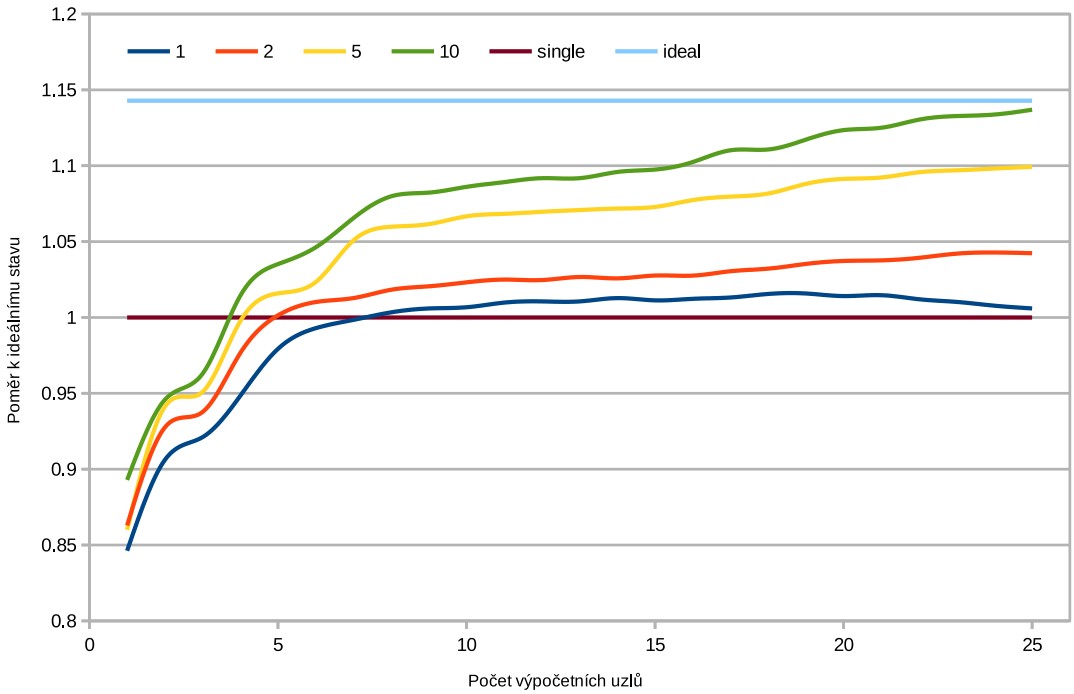
Čistá verze wrathionu na zvoleném virtuálním stroji dosáhla měřeného výkonu přibližně 40000 hesel za vteřinu, distribuovaná varianta běžící na stejném zařízení při distribuci balíku hesel každou vteřinu pak zvádala lámat hesel 35000.

8.2.1 Škálovatelnost

Cílem měření je dokázat, že upravená varianta wrathionu využívající pro distribuci balíků hesel mezi jednotlivé uzly clusteru knihovnu OpenMPI je v praxi použitelná, a že její výkonnost se při vhodně zvolených parametrech spuštění limitně blíží výkonu, jakého by dosáhl násobek jednotlivých uzlů běžících samostatně. Další užitečnou informací získanou tímto experimentem je reálný průběh výkonnosti implementovaného řešení, a také hledání

[2] www.digitalocean.com/
[3] www.aws.amazon.com/
[4] www.ansible.com/
[5] www.saltstack.com/
[6] www.puppet.com/

limitů, které jsou předpokládány při menších časových intervalech přidělování hesel a vyšších počtech uzlů.



Obrázek 8.1: Průběh výkonnosti v závislosti na počtu uzlů a různých faktorech škálovatelnosti

V grafu 8.1 je zobrazena naměřená výkonnost, a to jako závislost poměrného výkonu v porovnání s ideálním průběhem násobku výkonu samotného uzlu na množství zapojených uzlů. Tento ideální průběh počítá se zmíněným výkonem 40000 hesel za vteřinu změřeném na neupraveném wrathionu, je vynesen do grafu jako křivka ideal, a kopíruje osu x, stejně jako křivka vynesená pod popiskem single, zobrazující výkon, který by představovala série zařízení spouštštějících samostatně distribuovanou verzi wratihonu.

Mimo tyto dvě zmíněné konstanty vidíme vyneseny čtyři křivky se značně podobným profilem průběhu, směrem od osy X nahoru se pak tedy jedná o různá nastavení faktoru škálovatelnosti a to v pořadí od hodnoty nejnižší po nejvyšší (1, 2, 5, 10). Při distribuci balíků hesel jež uzel zpracuje za jednu vteřinu se výkon i při větším počtu zařízení jen ztěží vyšplhá přes hranici označenou single, natož aby vůbec došlo k přiblížení se k průběhu ideálnímu. Tento stav nastává kvůli režii spojené s komunikací, z grafu je taktéž patrné, že při přesáhnutí hranice přibližně dvaceti uzlů, začne naopak docházet k poklesu výkonu na uzel, znamená to tedy, že komunikace a čas strávený v její obsluze a obsluze kritických sekcí přesáhl únosnou mez. Vzhledem k tendenci kterou křivka výkonnosti nabírá v prostoru mezi 20 a 25 výpočetními uzly se tak dá předpokládat její postupná degradace.

Druhou diskutovanou křivkou je pak průběh při použití faktoru škálovatelnosti 10, tedy přiřazování balíků hesel jež vystačí zařízení po dobu deseti vteřin. Výkonnost samostatného MPI uzlu je překonána již při čtyřech zařízeních, a rychlost zkoušení generovaných hesel se při 25 uzlech již hraničně blíží ideálnímu průběhu, a tedy výkonnost clusteru jako celku je téměř rovna výkonnosti jeho jednotlivých uzlů, s výhodou absence nutnosti spouštět lámání

hesel na každém zařízení zvlášť.

U všech zvolitelných hodnot faktoru škálovatelnosti se dá předpokládat podobný průběh jako u varianty s jednou vteřinou, tedy pozvolný nárust výkonnosti až do hraničního stavu kdy množství komunikace začne způsobovat prostoje ve výpočtu a následné klesání. Čím větší hodnota faktoru škálovatelnosti, tím větší počet uzlů také bude potřeba pro dosažení této hranice, například u desetivteřinových lámacích oken se dá předpokládat odklon od křivky až u několika desítek až stovek zařízení.

Z pohledu na graf 8.1 je tedy zřejmé, že srovnatelné výkonnosti dosahuje OpenMPI varianta až při zvolení větších balíků hesel pro jednotlivá zařízení a při jejich větším množství. Toto však neznamená nezbytně větší výhodnost použití samostatně běžících aplikací pro obnovu hesel u několika různých souborů. Zvolením ještě větších faktorů škálovatelnosti totiž dochází k většímu rozdělení prostoru hesel a tedy vhodně spuštěná zeserializovaná OpenMPI verze Wrathionu velice pravděpodobně dosáhne nalezení všech hesel dříve, neboť v případě stejné rychlosti je nalezení za stejný čas logicky nejhorší stav, který by mohl nastat.

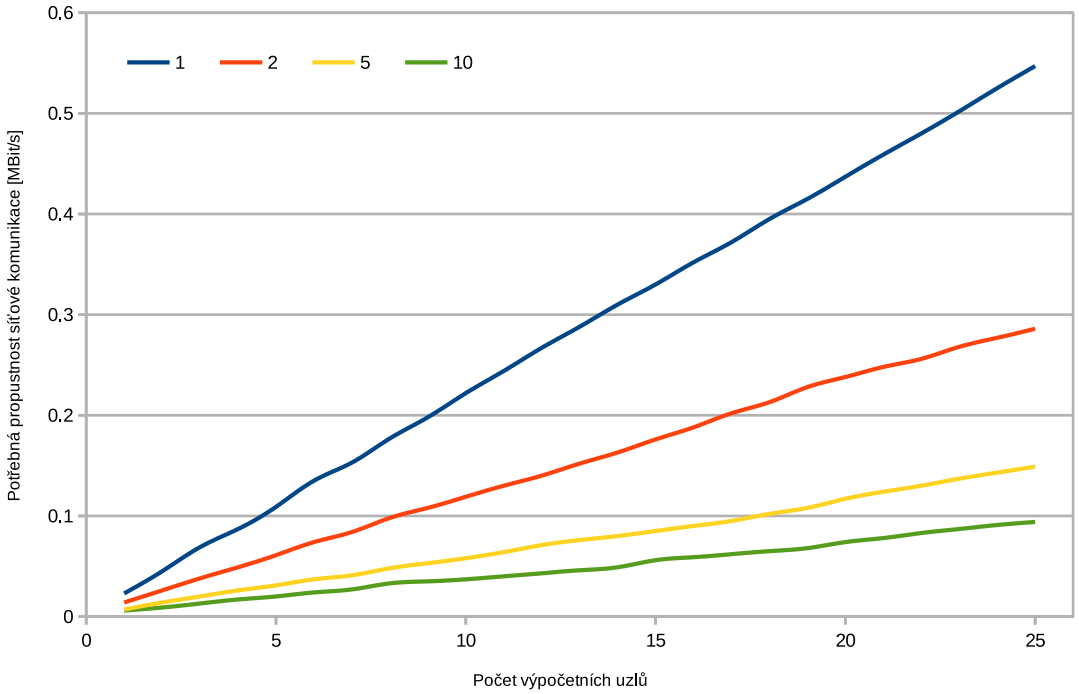
8.2.2 Zátěž síťového provozu

Poměrně důležitou částí experimentu, který se zabývá prováděním výpočtů nad větším množstvím zařízení, která jsou propojena pouze na úrovni sítě a nikoliv vysokorychlostní sběrnice, je i měření výkonu právě na této vrstvě. Pro MPI existuje řada nástrojů na profiling, za zmínku stojí již nevyvíjené MPE[7] běžící nad spouštěnou úlohou poskytující i grafický nástroj pro jednodušší analýzu komunikace mezi zařízeními která je trasována za jejich běhu, či konzolové mpiP[8] pracující pouze se statistickými informacemi. Tyto nástroje však neposkytují jednoduchou analýzu síťového provozu, nabízí se tak vytvoření záznamu komunikace pomocí utility `tcpdump` a její následná analýza nástrojem Wireshark[9]. Ten naneštěstí v základu nerozezná komunikaci protokolu MPI, k dispozici je však neoficiální plugin poskytující identifikaci a základní analýzu těchto paketů[10].

Předpoklad u měření vlastností komunikace je lineární průběh v závislosti na počtu uzlů, neznámou však je celkový datový tok. Při pohledu na funkce `MPI_Send` a `MPI_Recv` zdánlivě nepůjde o velké objemy dat, přenášeno je minimální množství, většina argumentů jsou proměnné typu integer, nejobjemnější částí může být přenášená hodnota, jde-li o řetězec či pole. Z dokumentací knihovny však není zřejmé zda, a případně rozsáhlost komunikace spojené se samotným během programu.

Na grafu 8.2 je vynesena závislost datového toku MPI provozu měřeného na hlavním uzlu na množství zapojených uzlů. Z tohoto je zřejmé, že přepoklad lineárního průběhu byl správný, navíc však byla experimentem zjištěna náročnost na vyžadovanou síťovou propustnost a z ní plynoucí jednoduchý výpočet, závislost je pouze na zvoleném parametru faktoru škálovatelnosti. Předpokládané nároky na každý uzel v závislosti na faktoru škálovatelnosti jsou uvedeny v tabulce 8.1. V grafu jsou opět uvedeny křivky pro hodnoty škálovacího faktoru 1, 2, 5 a 10, kdy nejvyšší z hodnot je vynesena nejblíže ose X a nejvzdálenější a nejstrmější je průběh pro faktor škálovatelnosti 1. Nároky jsou však na možnosti které poskytují dnešní technologie velice malé, 0,3MBit/s provoz při přiřazování balíků na dvě vteřiny a pětadvaceti uzlech nevytváří reálný základ pro používání speciálních zařízení.

[7] www.mcs.anl.gov/research/projects/perfvis/
[8] www.mpip.sourceforge.net/
[9] www.wireshark.org/
[10] www.github.com/juhulian/mpi-dissector/



Obrázek 8.2: Využitý datový tok v závislosti na počtu uzlů a různých faktorech škálovatelnosti

faktor škálovatelnost	1	2	3	4	5	6	7	8	9	10
rychlost na uzel [kbit/s]	22.4	11.7	9.2	7.3	6.1	5.65	5.1	4.65	4.15	3.85

Tabulka 8.1: Potřebný datový průtok na uzel v závislosti na faktoru škálovatelnosti

Nejdůležitějším faktorem síťové vrstvy tak je co nejnižší možný round trip time[11] (RTT), jelikož komunikace vedená arbitrem (kapitola 7.1.1) a rezervačními servery zařízení (kapitola 7.1.2) je potvrzovaná, a vyšší RTT by mohl způsobovat snížení výkonnosti při čekání na nový balík hesel.

8.2.3 Finanční zátěž výpočtu hesla

Při použití procesorové varianty a ceny za hodinu běhu zvoleného virtuálního stroje v prostředí DigitalCloud (3 americké centy za hodinu) se dostaneme na hodnotu vyzkoušení jednoho hesla 2.803×10^{-8} amerického centu. V praxi je taková částka zanedbatelná, je však potřeba si uvědomit že pro nalezení desetiznakového hesla v sadě ASCII-32 (viz kapitolu 2.1) je potřeba vyzokušet téměř 50 trilionů hesel, což by ve zvoleném prostředí stálo přibližně deset milionů amerických dolarů. Tento přístup tak pro praxi není příliš použitelný, a rozhodně se jeví jako vhodnější varianta koupě výkonné grafické karty, například AMD Radeon R9 Fury X [1], jejíž pořizovací cena se rovná osmi stům dolarů a výpočetní rychlost oproti zvolenému cloudovému stroji více než stonásobná.

Pro realizaci výpočtů za pomocí akcelerace na GPU se nabízí relativní novinka na poli

[11] čas strávený mezi odesláním datového paketu a zpracováním potvrzení o jeho přijetí na vzdáleném uzlu

cloudových služeb, a to vzdálený přístup ke GPU. Tuto službu nabízí například Amazon Web Services, k dispozici jsou aktuálně dvě úrovně dedikovaného stroje, a to s jednou, nebo čtyřmi grafickými kartami, každá o 1536 CUDA jádrech[12]. V porovnání se zmiňovanou kartou AMD R9 je pak přibližně o 50% výkonnější, cena za pronájem takového zařízení však stoupá na 2.6 amerického dolaru za hodinu.

8.2.4 Distribuovaná GPU varianta

Úspěšnost distribuce hesel pro běh na CPU variantě byla demonstrována v kapitole 8.2.1, minimálně stejné výkonnostní křivky se tak předpokládají i v distribuovaném běhu na grafických kartách za použití více reálných strojů. Při testování této varianty však bylo dosaženo vnitřní limitace prostředí OpenMPI. Po spuštění démona ORTE (jak bylo zmíněno v kapitole 7.3.3) dochází k inicializaci komponenty OPAL, která vytváří vrstvu zajišťující přístup k datům skrze celé distribuované prostředí. Tato inicializace proběhne v pořádku, avšak veškeré datové operace probíhají na kořenovém uzlu prostředí, a to včetně překladu kernelů pro grafické karty, které centrální uzel bez jakýchkoliv obtíží dokáže adresovat. Komplikace nastává v okamžiku, kdy se komponenty ORTE a OPAL pokusí zprostředkovat přenos zkompilovaných kernelů na cílové uzly. Velikost souborů je totiž pro prostředí příliš veliká, a tato komunikace selže, neboť cílové uzly nejsou schopny celý soubor uložit. Tuto limitaci v kódu OpenMPI se naneštěstí nepodařilo lokalizovat a odstranit. V případě, že uzel přistupuje k datům sám na sobě (spuštění distribuované varianty na jednom zařízení), dojde k bezproblémovému nahrání kernelu. Z tohoto faktu lze vyvodit závěr, že v případě že bude limitace prostředí OpenMPI odstraněna, bude distribuovaná varianta Wrathionu fungovat i na grafických kartách.

Jako řešení této situace se nabízí návrh pouze komunikačního podprogramu, který by inicializoval Wrathion na každém z uzlů, a poté by pomocí nově implementovaného aplikačního rozhraní wrathionu tomuto předával řídící informace, vzhledem k tomu, že by ale šlo o další vrstvu mezi aplikacemi, dá se předpokládat horší výkonnost, a zamýšlené řešení se velice nápadně blíží modelu platformy BOINC, a předpokládané úsilí správné konfigurace BOINC serveru oproti vlastnímu návrhu a implementaci OpenMPI hubu je podstatně nižší.

8.3 Zhodnocení experimentů

Řešení distribuce hesel mezi jednotlivá zařízení pomocí prostředí OpenMPI se ukázalo jako jednoduchá a v případě více než dvaceti uzlů i efektivní varianta, nešťastným řešením je však limitace v komponentě OPAL a tedy momentální nemožnost provádět distribuovanou obnovu na grafických kartách. Několikrát byla také zmíněna platforma BOINC, která dokáže na distribučním uzlu překládat CUDA jádra a následně je distribuovat na zařízení zapojená do výpočtu, a umožňuje hlavně komunitní užití, které v případě OpenMPI řešení není reálné (provázání všech zařízení pomocí bezheslového ssh přístupu). V komunitní verzi je sice potřeba každý balík hesel vyzkoušet na více než jednom zařízení, pro případ, že dané zařízení je kompromitováno a neposkytuje pravdivé informace, avšak použijeme-li BOINC framework v uzavřeném a kontrolovaném systému, nemusíme na tyto scénáře brát ohled. I přesto, že byly naměřeny poměrně slibné výsledky, jeví se použití BOINC frameworku jako výhodnější varianta, převážně kvůli již existujícím obslužným a kontrolním nástrojům (omezení výkonu, příprava fronty, ...).

[12] https://aws.amazon.com/ec2/instance-types/

8.4 Další možnosti rozšíření

V aktuální podobě je Wrathion spustitelný buďto jako CPU nebo jako GPU varianta, v případě distribuovaného řešení se pak nabízí možnost upravit inicializaci na jednotlivých uzlech do podoby, kdy u každého ze zapojených je přesně určeno zda bude spouštět cracker na CPU, GPU, nebo i všech zařízeních najednou.

Čas nutný k nalezení hesla je i přes částečné rozdělení mezi jednotlivá zařízení vždy dosti podobný, protože generátor přes prostor hesel postupuje lineárně. Vzhledem k tomu, že je každé heslo jednoznačně určeno svým celočíselným indexem, a prostor všech hesel je omezen parametry použité abecedy a maximální délky hesla, nabízí se možnost tento prostor rozdělit na bloky o stejné velikosti, zavést jejich indexaci, a jednotlivým zařízením pak přidělovat balíky náhodných indexů (podle jejich výkonnosti). Zavedením náhodného faktoru tak může dojít k podstatně rychlejšímu nalezení některých hesel, stejně tak k pomalejšímu, podle zákonů pravděpodobnosti by se však v případě serializace úkonů měl průměrný čas pohybovat okolo průměrného času lineárního generátoru.

Pro zjednodušení práce v distribuovaném prostředí bez použití nástrojů typu puppet či salt vzniká prostor pro vytvoření a používání nástroje který pomůže se správným vytvořením ssh vazeb mezi jednotlivými uzly a přípravu, případně kontrolu vytvořených souborů s mapováním zařízení a uzly pro argument utility mpirun.

V případě komplexnějších síťových architektur prostředí užívaných pro obnovu hesel se nabízí možnost instanciace centrálního arbitra a následně prioritnějších uzlů, které budou distribuovat data na uzly podřízené (stromová struktura), v takovémto případě by se v celém systémů nacházelo více objektů arbitra, centrální uzel by však byl autoritativní a přerozděloval balíky hesel pouze dalším arbitrům, nikoliv koncovým uzlům, které by si vyžádaly balíky hesel až od arbitrů podřízených hlavnímu.

Kapitola 9

Závěr

V práci byla vysvětlena problematika vyhledávacích tabulek, duhových tabulek a dalších možností, jak provést útok na individuální heslo. Také byly popsány metody obrany proti těmto útokům a to konkrétně přidávání soli a pepře při hashování hesla a vysoké nebo proměnné počty průchodů hashovacích funkcí. Z oblasti akcelerace výpočtů na grafické kartě byly představeny technologie OpenCL a CUDA a uvedeny jejich základní výhody a nevýhody. Zmíněna je řada prostředí používaných pro výpočty v distribuovaném počítačovém prostředí a provedeno zhodnocení vhodnosti na jednotlivé typy úloh. Popsán je nástroj pro lámání hesel na grafických kartách - Wrathion, včetně diskuze na téma dalších možných rozšíření. Z nastřádaných znalostí byl poté vytvořen a popsán návrh realizace rozšíření Wrathionu pro práci v distribuovaném prostředí, včetně topologických možností a přístupu k distribuci rozsahů hesel k lámání mezi jednotlivé výpočetní uzly.

Další část práce poté popisuje implementační detaily zvoleného návrhu, generátorová část kódu byla rozšířena o heslový arbitr a rezervační server zařízení, které spravují samotné přidělování hesel, hlavní vlákno Wrathionu pak doznalo největších změn v podobě rozdělení na běh hlavního a slave zařízení. Implementované změny byly podrobeny experimentům, v nichž se prokázalo, že při použití správných parametrů spuštění a dostatečného množství zařízení může být OpenMPI varianta Wrathionu efektivní jako samostatně spuštěná zařízení. Experimenty byla také zjištěna relativně nízká síťová náročnost OpenMPI řešení, kdy z tohoto pohledu jsou pak největší nároky kladeny na odezvu a ne rychlost.

Diskutována jsou také nabízející-se rozšíření distribuované varianty. Mezi ně můžeme zařadit hybridní variantu běhu s rozdělením jednotlivých využitých zařízení na CPU a GPU uzly. Dále pak nástroj generující mapování a provázání jednotlivých uzlů pomocí SSH spojení, a možnost úpravy distribuovaného řešení na víceúrovňové, kdy by role arbitra byla rozdělena mezi několik dalších zařízení mimo centrálního uzlu.

Literatura

[1] Advanced Micro Devices: *AMD Radeon R9 Fury X Graphics Card User Guide*. Advanced Micro Devices, Sunnyvale, Kalifornie, USA, 2015, [online; navštíveno 23.05.2016].
URL http://support.amd.com/Documents/amd-radeon-r9-fury-x.pdf

[2] Amazon Web Services: Linux GPU Instances. 2013, [online; navštíveno 23.05.2016].
URL http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using_cluster_computing.html

[3] Apostal, D.; Foerster, K.; Chatterjee, A.; aj.: Password recovery using MPI and CUDA. In *High Performance Computing (HiPC), 2012 19th International Conference on*, Prosinec 2012, s. 1–9, doi:10.1109/HiPC.2012.6507505.

[4] Beckman, K.: Hacking BitLocker and how to prevent unauthorized access to protected drives. 2015, [online; navštíveno 23.05.2016].
URL https://4sysops.com/archives/hacking-bitlocker-and-how-to-prevent-unauthorized-access-to-protected-drives/

[5] Faber, P.; Größlinger, A.: A Comparison of GPGPU Computing Frameworks on Embedded Systems. *IFAC-PapersOnLine*, ročník vol. 48, č. issue 4, 2015: s. 240–245, ISSN 24058963, doi:10.1016/j.ifacol.2015.07.040.
URL http://linkinghub.elsevier.com/retrieve/pii/S2405896315008150

[6] Focht, E.; Močnik, J.; Unger, F.; aj.: Light-Weight Kernel with Portals. In *High Performance Computing on Vector Systems 2010*, Springer Science Business Media, 2010, s. 3–16, doi:10.1007/978-3-642-11851-7_1.
URL http://dx.doi.org/10.1007/978-3-642-11851-7_1

[7] Graham, R. L.; Woodall, T. S.; Squyres, J. M.: Open MPI: A Flexible High Performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, Září 2005.

[8] Hranický, R., Matoušek, P., Ryšavý, O., and Veselý, V.: Experimental Evaluation of Password Recovery in Encrypted Documents. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*, Faculty of Information Technology, Brno University of Technology, 2016, ISBN 978-989-758-167-0, s. 299–306.

[9] Intel Corporation: *Intel Xeon Processor E5-2650L v3*. Mountain View, Kalifornie, USA, 2014, [online; navštíveno 23.05.2016].
URL http://ark.intel.com/products/81903/Intel-Xeon-Processor-E5-2650L-v3-30M-Cache-1_80-GHz

[10] Intel Corporation: Intel Scalable System Framework. 2015, [online; navštíveno 23.05.2016].
URL http://www.intel.com/content/www/us/en/high-performance-computing/product-solutions.html

[11] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual*. Mountain View, Kalifornie, USA, Duben 2016, [online; navštíveno 23.05.2016].
URL http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[12] Karimi, K.; Dickson, N. G.; Hamze, F.: A Performance Comparison of CUDA and OpenCL. *CoRR*, ročník abs/1005.2581, 2010, [online; navštíveno 23.05.2016].
URL http://arxiv.org/abs/1005.2581

[13] LastPass: Password Iterations (PBKDF2). 2016, [online; navštíveno 23.05.2016].
URL https://helpdesk.lastpass.com/account-settings/general/password-iterations-pbkdf2/

[14] Message Passing Interface Forum: *MPI: A Message Passing Interface Standard Version 2.2*. Message Passing Interface Forum, Září 2009, [online; navštíveno 23.05.2016].
URL https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[15] Microsoft: BitLocker Drive Encryption Overview. 2007, [online; navštíveno 23.05.2016].
URL http://windows.microsoft.com/en-us/windows-vista/bitlocker-drive-encryption-overview

[16] Microsoft: What is Encrypting File System (EFS). 2012, [online; navštíveno 23.05.2016].
URL http://windows.microsoft.com/en-us/windows/what-is-encrypting-file-system

[17] National Institute of Standards and Technology: *Secure Hash Standard (SHS)*. 2015, [online; navštíveno 23.05.2016].
URL http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[18] NVIDIA: *CUDA Toolkit Documentation*. Santa Clara, Kalifornie, USA, 2015, [online; navštíveno 23.05.2016].
URL http://docs.nvidia.com/cuda/cuda-binary-utilities/#maxwell

[19] NVIDIA: GPU Applications - ANSYS. 2016, [online; navštíveno 23.05.2016].
URL http://www.nvidia.com/object/tesla-ansys-accelerations.html

[20] Open Web Application Security Project: Blocking Brute Force Attacks. 2016, [online; navštíveno 23.05.2016].
URL https://www.owasp.org/index.php/Blocking_Brute_Force_Attacks

[21] prweb: ElcomSoft Breaks Wi-Fi Encryption Faster with GPU Acceleration. Říjen 2015, [online; navštíveno 23.05.2016].
URL http://www.prweb.com/releases/wi-fi/cracking/prweb1405954.htm

[22] RenderLab: Church of Wifi WPA-PSK Lookup Tables. [online; navštíveno 23.05.2016].
URL http://www.renderlab.net/projects/WPA-tables/

[23] SanDisk Corporation: SanDisk introduces security-certified self-encrypting solid state drive for corporate environments. 2014, [online; navštíveno 23.05.2016].
URL https://www.sandisk.com/about/media-center/press-releases/2014/sandisk-introduces-security-certified-self-encrypting-solid-state-drive-for-corporate-environments

[24] Schmied, J.: *GPU Akcelerované prolamování šifer*. Diplomová práce, Fakulta Informačních Technologií, Vysoké Učení Technické v Brně, 2014.

[25] Schneier, B.: Secrecy, Security, and Obscurity. Květen 2015, [online; navštíveno 23.05.2016].
URL https://www.schneier.com/crypto-gram/archives/2002/0515.html#1

[26] Sharpened Productions: The Slangit Leet Speak. 2016, [online; navštíveno 23.05.2016].
URL http://slangit.com/leet_sheet

[27] Shimpi, A. L.; Wilson, D.: GPU Transcoding Throwdown: Elemental's Badaboom vs. AMD's Avivo Video Converter. Prosinec 2015, [online; navštíveno 23.05.2016].
URL http://www.anandtech.com/show/2685

Přílohy

Seznam příloh

A Obsah CD **48**

Příloha A

Obsah CD

- `src` - zdrojové kódy aplikace
- `doc` - text diplomové práce
- `tex` - zdrojové soubory diplomové práce
- `README` - pokyny pro práci s aplikací